Edward Mauricio Alférez Salinas

Derivation and Consistency Checking of Models in Early Software Product Line Engineering

Dissertação para obtenção do Grau de Doutor em
Engenharia Informática

Orientadora : Prof. Dra. Ana Maria Diniz Moreira,
Associate professor, Universidade Nova de Lisboa

Co-orientador : Prof. Dr. Vasco Miguel Moreira do Amaral,
Assistant professor, Universidade Nova de Lisboa

Júri:

Presidente: Prof. Dr. Luís Manuel Marques da Costa Caires

Arguentes: Prof. Dr. Xavier Franch
Prof. Dr. Jean-Michel Bruel

Vogais: Prof. Dr. António Rito Silva
Prof. Dr. Ademar Manuel Teixeira de Aguiar
Prof. Dr. João Batista da Silva Araújo Júnior
Prof. Dra. Ana Maria Diniz Moreira
Prof. Dr. Vasco Miguel Moreira do Amaral

FACULDADE DE CIÊNCIAS E TECNOLOGIA
UNIVERSIDADE NOVA DE LISBOA

Dezembro, 2012

Derivation and Consistency Checking of Models in Early Software Product Line Engineering

a mi amada familia

Acknowledgements

The path to pursue a Ph.D. is not only embellished of fresh ideas, sudden inspiration and scientific beauty; it is often a path with many obstacles and throwbacks. However, thanks to God I was able to make my way regardless all the difficulties.

Also, I want to thank special people that accompanied and supported me. First, I want to thank to my parents who supported me, my dreams, ideas, and plans from the very beginning of my life. Furthermore, I want to acknowledge my brother that believed in me, and that together with my parents encouraged me to start and finish this Ph.D. project.

In my life I had many teachers, mentors, and professors – too many to mention here. Certainly, my advisers played a key role in my dissertation. It is worth noting that I had not just one adviser, but two advisers who supported me in different ways. First, Ana Moreira helped me to refine my style in scientific thinking, working, and writing. She called me to work in her team in Portugal for a European project and next gave me the opportunity for doing my Ph.D. Second, I want to thank my co-advisor Vasco Amaral who believed in my abilities and supported me unconditionally. I learned many invaluable truths from him about the world of research, science, and academics. I thank to Ana and Vasco for every discussion about fundamental issues of my research. It was an honour and a real pleasure working with them. Their support, encouragement, and advice have gone further than I would have imagined and expected.

As further mentors I want to thank Marta Silvia Tabares and Raquel Anaya. I met and worked with them before starting my Ph.D. work. I profited always from our discussions and useful advice that helped me to believe in research and to continue my studies in Europe. During the years of my Ph.D. studies I worked and discussed with many other researchers that contributed to the evolution of my thinking and understanding of many problems in computer science. Apart from my advisers and mentors some of the most influential persons were João Araújo, Uirá Kulesza and Roberto Lopez-Herrejon.

A special thank-you goes to the Centro de Informática e Tecnologias de Informação (CITI), Portugal, the Departamento de Informática of the Universidade Nova de Lisboa, the European Project AMPLE, contract IST-33710, and the Fundação para a Ciência e a Tecnologia that through the grant SFRH/BD/46194/2008 supported me financially and

organizationally over all this time.

Lastly, I offer my regards and blessings to all my friends and relatives that supported me in any respect during this period of my life.

Abstract

Software Product Line Engineering (SPLE) should offer the ability to express the derivation of product-specific assets, while checking for their consistency. The derivation of product-specific assets is possible using general-purpose programming languages in combination with techniques such as conditional compilation and code generation. On the other hand, consistency checking can be achieved through consistency rules in the form of architectural and design guidelines, programming conventions and well-formedness rules. Current approaches present four shortcomings: (1) focus on code derivation only, (2) ignore consistency problems between the variability model and other complementary specification models used in early SPLE, (3) force developers to learn new, difficult to master, languages to encode the derivation of assets, and (4) offer no tool support.

This dissertation presents solutions that contribute to tackle these four shortcomings. These solutions are integrated in the approach Derivation and Consistency Checking of models in early SPLE (DCC4SPL) and its corresponding tool support.

The two main components of our approach are the Variability Modelling Language for Requirements (VML4RE), a domain-specific language and derivation infrastructure, and the Variability Consistency Checker (VCC), a verification technique and tool. We validate DCC4SPL demonstrating that it is appropriate to find inconsistencies in early SPL model-based specifications and to specify the derivation of product-specific models.

Keywords: Software Engineering, Software Product Line Engineering, Model-Driven Development, Domain-Specific Languages Engineering, Software Verification, Modelling.

Resumo

Engenharia de Linhas de Produtos de Software (ou SPLE, do inglês *Software Product Line Engineering*) deve ser capaz de expressar a derivação de artefactos para produtos específicos, ao mesmo tempo que verifica a sua consistência. A derivação de artefactos para produtos específicos é possível usando linguagens de programação de propósito geral em combinação com técnicas como compilação condicional e geração de código. Por outro lado, a verificação de consistência pode ser alcançada através de regras de consistência na forma de diretrizes de desenho e arquitetura, convenções de programação e regras de boa formação. As abordagens atuais apresentam quatro insuficiências: (1) concentram-se apenas na derivação de código, (2) ignoram problemas de consistência entre o modelo de variabilidade e outros modelos de especificação complementares utilizados nas fases iniciais do SPLE, (3) forçam aos desenvolvedores a aprender novas linguagens, difíceis de dominar, para codificar a derivação de artefactos, e (4) não oferecem suporte ferramental.

Esta dissertação apresenta soluções que contribuem para enfrentar estas quatro insuficiências. Estas soluções são integradas na abordagem DCC4SPL (do inglês *Derivation and Consistency Checking of models in early SPLE*) e suas ferramentas de suporte.

As duas principais componentes da nossa abordagem são o VML4RE (do inglês *Variability Modelling Language for Requirements*), uma linguagem de domínio específico e infraestrutura derivação, e o VCC (do inglês *Variability Consistency Checking*), uma técnica e ferramenta de verificação. A abordagem foi validada, demonstrando que o DCC4SPL é apropriado para identificar inconsistências nas especificações de Linhas de Produtos de Software baseadas em modelos e para especificar a derivação de modelos para produtos específicos.

Palavras-chave: Engenharia de Software, Engenharia de Linha de Produto de Software, Desenvolvimento Dirigido por Modelos, Engenharia de Linguagens de Domínio Específico, Verificação de Software, Modelação.

Contents

List of Figures

List of Tables

Abbreviations

AMPLE	Aspect-Oriented, Model-Driven Product Line Engineering
AOM	Aspect-Oriented Modelling
AOSD	Aspect-Oriented Software Development
CCCMS	Car Crash Crisis Management System
CNF	Conjunctive Normal Form
DCC4SPL	Derivation and Consistency Checking of models in early SPLE
DSL	Domain Specific Language
DSLE	Domain Specific Language Engineering
EBNF	Extended Backus–Naur Form
Ecore	Eclipse Modelling Framework Core
EMF	Eclipse Modelling Framework
HUTN	Human Usable Textual Notation
MDD	Model-Driven Development
MOF	Meta-Object Facility
OMG	Object Management Group
OOP	Object-Oriented Programming
SPL	Software Product Line
SPLE	Software Product Line Engineering
UML	Unified Modelling Language
VCC	Variability Consistency Checker
VML4RE	Variability Modelling Language for Requirements

Part I

Overview

1

Introduction

The current trend of globalization is pressuring organizations developing software-intensive products to explore efficient ways to provide high-quality products of increasing size and complexity customized to the special needs of individual customers of market segments. Software Product Line Engineering (SPLE) [79, 27, 31] is a software development paradigm that aims at addressing this challenge based on the observation of three facts: (1) most products in a market segment or application domain are not new, (2) products usually share many common features, and (3) most organizations build software systems in a particular domain, repeatedly releasing product variants by removing features or adding new features. SPLE takes into account these three insights to provide an approach to increase the productivity of software development organizations, enabling them to cope with the diversity of the global market to develop product variants through systematic reuse of software assets which have been proactively planned with respect to expected future requirements [86].

During the first years of SPLE, attention has been focused on detailed design and code generation for new products. To date there is no effective approach to deal with the issue of derivation and consistency checking of models during early modelling of Software Product Lines (SPLs). Early modelling is requirements specification and architectural design. This part of the software development process eases reasoning about the products to be derived, establishes critical trade-offs early in the software life cycle, and supports the creation of software specifications for the developers before implementation in code (where changes may be difficult and expensive to make).

This dissertation addresses the issue of derivation and consistency checking of models with a focus on early modelling. To ease the derivation of products, which should be automated as much as possible, it is not enough to collect software assets such as

requirements and architectural models in a repository, expecting that they will be reused for the creation of new products. Instead, it is required to describe what assets are available, what variable and common features the assets have, which features will be part of each product, and how each selection of features triggers the transformation of reusable assets to create concrete software products. The way these descriptions are made, used, and processed represents a significant research challenge due to the difficulty to modularize variabilities and to express and derive concrete products at the level of abstraction that the modellers require.

Products derivation cannot be addressed in isolation because the quality of the products is also determined by the consistency between the feature model (an SPL variability model) and other assets that model the features. The problem of deriving products using inconsistent inputs is that it produces products that do not satisfy their requirements, therefore of low-quality and less cost-effective (due to the effort and time to find inconsistencies and repair their effects during code derivation). In the context of this dissertation, inconsistencies are violations of consistency rules (architectural and design guidelines, programming conventions and well-formedness rules) established between a variability model and other models. Inconsistencies are rarely simple in practise because they may involve several interrelated features and model fragments in different models which are usually developed by diverse stakeholders.

In addition to the derivation of products and consistency checking, a common factor that adds complexity to the development of SPLs is that there is not one customer or company supplying the requirements. In fact there are potentially hundreds of yet unknown customers which make the number of features, assets, and their incompatibilities and dependencies to rise exponentially. To derive products or to check consistency given a large number of combinations of features is not feasible without an adequate approach and proper automation support.

This dissertation addresses models derivation and consistency checking by proposing a novel and tool-supported approach called Derivation and Consistency Checking of models in early SPLE (DCC4SPL). One of the main characteristics of DCC4SPL is that it employs models that help developers to work at the level of abstraction that they require. This right level of abstraction is reached by hiding or masking implementation details, using a specification vocabulary familiar to system modellers, bringing out the big picture, or focusing on different aspects of a system. The use of models favours the application of Model-Driven Development (MDD) [94, 85] where models are the primary development assets, and concrete systems are produced as a result of model transformations. Thus, DCC4SPL uses SPL assets as input models that are transformed to derive customized models expressing product requirements specifications or architectural models. Given that the languages used to write the models have a well-defined form (syntax) and meaning, DCC4SPL automates the mining and processing of dependencies and incompatibilities the variability model and other models to check consistency of the SPL specification.

The two main parts of DCC4SPL are models derivation and consistency checking.

Models derivation is supported by VML4RE. VML4RE is a Domain Specific Language (DSL) that specifies how to derive models for specific products based on the transformation of reusable model fragments. This language eases the derivation of models by specifying usually complex model transformations employing a vocabulary familiar to requirements engineers and software architects. VML4RE is inspired in the Aspect-Oriented paradigm [44, 19] and, therefore, it refers to the process of derivation of models as a composition process designed to weave different models and model fragments related to several systems concerns (each concern corresponds to a feature or groups of features in the SPL context) to finally produce the models for concrete systems.

The second part of DCC4SPL is VCC, which addresses consistency checking between feature models and other models that design the features, for example: activity and component diagrams. VCC mines relationships between features from the feature model and from other models that design the features. Then, it employs propositional formulas to relate all the mined relationships. Checking if all the products in an SPL satisfy consistency rules is based on searching for a satisfying assignment of a propositional formula. The novelty of VCC, is fourfold: (1) it can be applied to early model-based SPLE such as requirements specification and architectural design, (2) it considers complex composition situations where the customization of models for specific products depends on the presence or absence of sets of features (or any other variability unit), (3) it checks consistency for the entire SPL instead of individual products, without compromising efficiency, and (4) since our focus is on early modelling, VCC does not assume a correct and complete feature model; instead, it helps developers to complete the feature model based on the detected inconsistencies with respect to other models.

As noted at the beginning of this section, one of the important objectives of SPLE is the efficient derivation of high-quality products. While the results related to the achievement of these challenges demonstrate the value of DCC4SPL, the organizational and economical aspects also determine the success in SPLE but are out of the scope of this dissertation. We believe that no single tool or approach can provide the optimal and general solution for any software engineering problem. Instead, the contributions of this dissertation provide significant pieces in SPLE and its technology tool box that consist of the tool-supported approach DCC4SPL.

1.1 Research Question

Based on the need to support the derivation of particular products from reusable assets, and the need to check consistency between the SPL feature model and the assets that are modelling its features, the main research statement of our work is to investigate:

How to guarantee effective consistency checking and derivation of product-specific models in early Software Product Line Engineering?

To understand the meaning of this research question we will analyse it into its six major keywords[1]:

- *Guarantee* means "to assure a particular outcome"[2]. In the context of this dissertation the main outcome is the derivation of products and consistent specification in early modelling (i.e., requirements and architecture modelling). This outcome includes two parts: (1) specifications on how to derive product-specific early models (or *model-based specifications*) expressed at the level of abstraction that developers require, and (2) the guarantee that the early models and the feature model of the SPL are consistent between them. (These will be explained further in Section 1.2 - Research Topics)

- *Effective* means two things: first, "producing or capable of producing an intended result" and second, "presently existing in fact and not merely potential or possible"[3]. This means that apart from providing a conceptual base and process we are also interested in a prototypal tool support that concretizes them.

- *Consistency* means "ability to be asserted together without contradiction" and "agreement or harmony of parts or features to one another or a whole"[4]. Inconsistencies result from the violations of consistency conditions that can be as diverse as architectural and design guidelines, programming conventions and well-formedness rules. In the context of this dissertation, consistency checking implies the verification of consistency conditions related to the relationships between the feature model and its related model-based specifications. Such relationships may be dependencies and incompatibilities between parts of models, and dependencies and incompatibilities between features.

- *Derivation* means "the act of obtaining something from a source or origin"[5]. In the context of this dissertation, derivation of products means to transform requirements and architectural models to obtain product-specific requirements and architectural models. In Aspect-Orientation it is common to use the term "composition" to refer the idea of "bringing together separately created software elements" [44]. Given that the VML4RE tool that supports the derivation of products is inspired in the Aspect-Orientation paradigm, we usually employ the term "composition" to refer to the derivation process. We use model transformations to actually perform the composition. Model transformations allow adding, removing, connecting, or replacing parts of models with new or already existing parts of other models.

- *Early Modelling* means to specify system concerns at the requirements specification and architectural design development stages. Examples of models used during early

[1] Some of them will be further discussed during Chapter 2 - Background.
[2] http://www.thefreedictionary.com/guarantee
[3] http://www.thefreedictionary.com/effective
[4] http://www.merriam-webster.com/dictionary/consistency
[5] http://www.thefreedictionary.com/derivation

modelling of software are use case, activity and components diagrams written in the Unified Modelling Language (UML). In a model-driven perspective of software development the use of models provide a vehicle to direct the course of understanding, design, construct, deploy, operate, maintain and modify software [70].

- *Software Product Line* means "a set of software-intensive systems that share a common, managed set of features satisfying the specific needs of a particular market segment or mission and that are developed from a common set of core assets in a prescribed way" [27]. We provide background information about software product line engineering in Subsection 2.3.1 - Software Product Line Engineering (SPLE).

The previous six keywords that compose our main research question establish the central goal of this dissertation: to provide an approach with tool support that guarantees the derivation of product-specific early models assuring that what it is supposed that the SPL can produce (expressed in a variability model such as a feature model) does not contradict with the definition of what can be actually produced (expressed with early models).

1.2 Research Topics

The research question sets the context of our research agenda, which we decomposed into two finer-grained, more focused topics that constitute the main concerns of this dissertation:

1. **Support to express and perform product-specific models derivation**. General-purpose model transformation languages, such as ATL [52], AGG [89], QVT [6] [75] require that developers have a deep knowledge of the abstract syntax of the models to describe their composition. However, most developers do not know the details of the abstract syntax of the languages that they use [51]. Also, even if developers know those details, there is still a barrier of learning and applying correctly the specifities (e.g., syntax and computing style) of new highly specialized languages such as model transformations languages. This research topic leads to the following research question:

   Question 1. How to support expressing and perfoming product-specific model derivation?

2. **Support for consistency checking between variability model and other models**. To produce product-specific models from an SPL, models have to be composed according to a specific selection of variability units (e.g., features) from a variability model (e.g., a feature model). This process requires a mapping between variability units, and artefacts, such as models, that specify them. A number of different approaches have been proposed to create mappings among variability models

[6] http://www.eclipse.org/projects/project.php?id=modeling.m2m.qvt-oml

and other models [48, 31]. However, ensuring consistency between variability models and not code-based assets has not been thoroughly researched. Consistency checking is particularly important given the need to assure that variability and models' constraints are reliable. Failures in consistency will produce customized systems that do not satisfy their requirements. This research topic leads to the following research question:

> *Question 2. How to support consistency checking between the variability model and other models?*

The definition of these two research topics was necessary to set the boundaries and the focus of the research presented in this dissertation. The results of the research and development on these two research topics, guided by Question 1 and Question 2, lead to the major contributions of this Ph.D. dissertation.

1.3 Contribution Overview

The three main contributions of DCC4SPL organized by research question are as follow:

1. **Model derivation using VML4RE.** The research question 1: *"How to support expressing and performing product-specific model derivation?"* is addressed with the Variability Modelling Language for Requirements (VML4RE). VML4RE is both a domain specific language and derivation infrastructure specifically tailored to express how to derive product-specific requirements models. We provided a specialization of VML4RE to model use scenarios as it provides language constructs (e.g., actions[7]) specifically designed for compositions of UML use cases and activity diagrams.
   VML4RE was one of the base languages used to create Variability Modelling Language (VML*) which was a common contribution created as a collaboration between the partners of the research project AMPLE (Section 1.5 - Research Context). VML* is used as an infrastructure to support the creation of new languages similar to VML4RE, each one specialized in a different kind of models. For example a version for architecture called VML4Arch was developed [97]. VML4RE presents the following benefits:

   - *Vocabulary familiarity.* It supports the process to specify the derivation of product-specific models, using a vocabulary (i.e., syntax) and concepts familiar to SPL developers. This hides the details of the model transformation language and process from the SPL developers. Thus, helping them to avoid thinking about many of the implementation details of general purpose model transformation languages.

[7] *Action* is the term used to refer to calls to model transformations written using a language familiar to developers.

- *Derivation flexibility.* It provides actions types for both positive[8] and negative[9] models' derivation mechanims.
- *Modularity.* It supports a better modularization of feature specifications. It allows to model each feature, or group of closely related features, in separated models. Also, it separates variability information (i.e., configuration knowledge[10]), instead of integrating it directly into the SPL models as other authors propose (e.g., [30, 45, 25]). Modularity to express feature specifications and variability information leads to better stability of the early models specifications, composition specifications and configuration knowledge.

2. **Consistency checking between variability model and other models using VCC**. The research question 2: *"How to support consistency checking between the variability model and other models?"* is addressed with the Variability Consistency Checking (VCC). VCC is a verification approach and tool that supports consistency checking between a variability model (e.g., a feature model), and the models that design the variability units of the variability model (e.g., features). There are different kinds of models used to design (sometimes we say, *realize*) the meaning of variability units, for example: use case, activity, and component diagrams.
   VCC mines constraints between variability units from the variability model and from the models that design variability. Then, VCC employs propositional formulas to relate all the mined constraints. Checking if all the products in an SPL satisfy consistency constraints is based on searching for a satisfying assignment of a propositional formula. VCC also presents the elements involved in a violation of a consistency rule and explains the cause of the inconsistency. VCC presents the following benefits:

   - *Consistency checking genericity*. It can be applied to check consistency of any model. We show examples of features models and several types of UML models, such as use case, activity and component diagrams. Also, it considers complex composition situations where the derivation of models for specific products depends on the evaluation of variability unit expressions[11], and not only depending on the presence or absence of individual variability units.
   - *Consistency checking multi-view awareness.* It can check consistency between multiple models and the variability specification independently of the number of languages employed.

[8] Positive variability is a derivation mechanism where new models are created adding parts to an initial model.

[9] Negative variability is a derivation mechanism where new models are created removing parts from an initial model.

[10] Configuration knowledge is the mapping between features and model fragments and the information on how to derive product-specific models according to selections of features.

[11] Variability unit expressions (e.g., feature expressions) are represented as different kinds of propositional formulas: variability unit name, negation, conjunction and disjunction.

- *Consistency checking scalability.* It checks consistency of all possible products during domain engineering, without compromising the performance of the verification process.

3. **Tool support**. The tool support contributes both research topics: "support to express and perform product-specific models derivation" and "support for consistency checking between variability model and other models". We developed three tool prototypes:

   - *VML4RE*. VML4RE uses EMFTEXT[12] which provides the software infrastructure to derive a concrete syntax and Eclipse plug-in editor based on the metamodel of each language written in EMF[13] Ecore[14]. This technology separates concrete syntax and abstract syntax, easing maintenance of the languages. The style of the concrete syntax chosen for VML4RE is HUTN (Human Usable Textual Notation)[15] provided by EMFTEXT. This style and the VML4RE language constructs, provide a suitable notation for describing requirements composition.
   - *VCC*. This tool alleviates developers from the tedious and error-prone task of manually checking variability models and their related model-based specifications for consistency. VCC has several reusable components, for example to map variants and model elements, to assign and persist identifiers for variability units, to translate from propositional logic to Conjunctive Normal Form (CNF) form (a format readable by satisfiability solvers), and to create constraints based on composition specifications and model-based specifications.
   - *Feature model metamodel and editor*. This implementation is written in EMF Ecore. The metamodel was a translation to EMF Ecore of the features model implemented as a Java library by the Software Product Lines On-line Tools (SPLOT) research community. All our implementation is open-source and relased as two Eclipse plug-ins in the SPLOT website[16].

The contributions of this work have been published and presented in international conferences and in specialized book chapters (Part II - Research Papers includes a copy of the most relevant). The publications target the research topics and contribute to specific parts of the DCC4SPL approach (described in Chapter 3). Part II - Research Papers provides more information about each publication such as authors, summary, authors' contribution, publication event and references to workshops, technical reports and papers presented in journals and conferences that complement the main publications.

[12] http://www.emftext.org/: EMF textual concrete syntax mapper.
[13] www.eclipse.org/emf/: Eclipse Modeling Framework.
[14] www.eclipse.org/emf/: EMF Core.
[15] http://www.omg.org/spec/HUTN/: The HUTN specification.
[16] http://www.splot-research.org/: Software Product Line Online Tools.

Nº	Title of the Publication	Contrib.
A	A Model-Driven Approach for Software Product Lines Requirements Engineering [9]	1
B	Multi-View Composition Language for Software Product Line Requirements [15]	1, 3
C	VML* – A Family of Languages for Variability Management in Software Product Lines [97]	1, 3
D	Model-Driven Requirements Specification for Software Product Lines [13]	1
E	Evaluating Approaches for Specifying Software Product Line Use Scenarios [6]	1
F	Supporting Consistency Checking between Features and SPL Use Scenarios [12]	2, 3
G	Ensuring Consistency between Feature Models and Model-Based Specifications - The VCC Approach [11]	2, 3

Table 1.1: Publications mapped to their contributions.

Table 1.1 lists the publications and assigns to them an identifier to facilitate the description of each contribution and the reference of each publication. Table 1.1 also summarizes the relationships between contributions and publications. The research papers that contributed the most to define the specification of models derivation are papers A to E, and the research papers that contributed the most to define the consistency checking are papers F to G. Tool support is defined in several papers, from B to C, and from F to G.

1.4 Research Method

In this dissertation we based our research in the *Technology*[17] *Research* method. This method is suitable for the purpose of producing new or better artefacts [84]. Once new artefacts are ready, researchers have to show that they actually fulfil the posed requirements and thereby satisfying the need on which they are based. Technology research does not always produce artefacts that are complete, regarded from a user's point of view. It is common to make a so-called functional prototype, which must be able to undergo the necessary evaluation. If the prototype looks promising, it can later on be elaborated to a complete, commercial product. Such finalization is typically done by other people than researchers [84].

The main steps in the technology research process are:

- *Problem analysis* where researchers answer "what is the potential need" for a new technology. In this step researchers identify a need for new or better artefacts .
- *Innovation* where researchers answer "how to make an artefact that satisfies the need" identified during problem analysis. In this step researchers produce new or better artefacts.
- *Validation* where researchers answer "how to show that the artefact satisfies the need". In this step it is necessary to check that the artefacts created during innovation satisfy their requirements. When new artefacts are obtained, the researcher has a basis for

[17] Technology: "the knowledge of artefacts emphasizing their manufacturing" [84].

new questions, leading to new investigations. Therefore, technology research is an iterative process.

Next, for each step we outline a summary on how we followed the technology research process in this work:

1. Problem analysis. Based on the literature review and the analysis of existing approaches for SPLE, we found two problem areas which were introduced in Section 1.1: "support to express and perform product-specific models derivation", and "support for consistency checking between variability model and other models". We presented the literature review in a technical report of the state-of-the-art in model-driven SPL [60]. Also, the industrial partners of our European research project AMPLE[18], Siemens A.G, SAP A.G and Holos, helped to identify the problem areas addressed in this dissertation.

2. Innovation. The innovation phase tackled the problem areas identified during problem analysis with three contributions (described in Section 1.3 - Contribution Overview): "model derivation using VML4RE", "consistency checking between variability model and other models using VCC" and "tool support". The new technology represented by DCC4SPL, is described in the research papers in Part II - Research Papers, and in the Chapter 3 - DCC4SPL Approach of this dissertation.

3. Validation. The results were validated with literature search, examples and case studies, prototypes, comparative metrics, and feedback obtained during the elaboration and presentation of peer-reviewed scientific publications.

   - *Literature search* of existing techniques and tools has been applied related to all research results. A continuous process of reviewing other research results and technology developments has been undertaken.
   - *Examples and case studies* were established and reused to validate our approach and those of others. The main two case studies are the SPL Smart Home and the Car Crash Crisis Management System. Smart Home [71] was defined as a case study in the AMPLE project. The Car Crash Crisis Management system was proposed by a group of international researchers as a common case study to compare different modelling approaches [56]. Both case studies were used along this dissertation to illustrate DCC4SPL.
   - *Several prototypes* support this dissertation to illustrate and validate the concepts and languages defined. The main two prototypes are the domain specific language and derivation infrastructure VML4RE (Variability Modelling Language for Requirements) and the verification approach and tool VCC (Variability Consistency Checker) that checks consistency between feature models and other

[18] European project, www.ample-project.net

models. Also, we built a feature model metamodel and editor based on an existing feature modelling library provided by the SPLOT research community.

- *Comparative metrics* were used to validate and further strengthen the results. We apply quantitative and qualitative methods that gathered statistical data to support the research claims. For example, the paper "Evaluating Approaches for Specifying Software Product Line Use Scenarios" presents an empirical evaluation that takes into account relevant quality attributes for variability management (e.g., modularity, expressivity and stability). It compares four key, representative techniques that aim at a better management of common and variable model specifications between products of an SPL. The techniques chosen to be evaluated span from type of notation (graphical or text-based), style to resolve variability (based on annotations or compositions), and quantification expressiveness. The result of this study concluded that our approach improves both modularity of the specifications of features along the models as well as stability of the models. Also, Chapter 4 - Validation, employs more attributes and metrics.
- *Peer-reviewed scientific publications* were produced for our main contributions. Having our main results evaluated by international specialized researchers further strengthens the validity of our claims. A copy of the main papers is given the Part II - Research Papers of this dissertation.

We believe that validating our research results using case studies, examples, prototypes, comparative metrics, and peer-reviewed scientific publications, as well as the positive feedback of our academic and industrial partners in the European project AMPLE, shows the relevance of the results obtained.

1.5 Research Context

The research work leading to this dissertation started in the context of the European project AMPLE (Aspect-Oriented, Model-Driven Product Line Engineering). The aim of AMPLE was to provide an SPL development methodology to offer improved modularization of variations, their holistic treatment across the software lifecycle and maintenance of their traceability during SPL evolution. AMPLE combined AOSD (Aspect-Oriented Software Development) [44, 19] and MDD [58, 94, 85] techniques to both address variability at each stage in the SPLE lifecycle and manage variations in associated artefacts, from requirements through architecture to code.

Our team at the Universidade Nova de Lisboa was responsible for coordinating the work package dedicated to Requirements Engineering (WP1) and several other tasks in other work packages. The work conducted for this dissertation started mainly under WP1, focusing on variability modelling and composition. Consistency checking is a topic that we developed after the end of AMPLE mainly with researchers from the Institute for

Software Engineering and Automation (SEA) at the Johannes Kepler University, Austria[19]. Also, part of the evaluation of our approach related to variability modelling and composition was performed together with researchers from the Software Productivity Group (SPG), Brazil. SPG is a multi-institutional group formed by researchers from the Informatics Center of the Federal University of Pernambuco (CIn/UFPE)[20], the Department of Computing Systems of the Federal University of Campina Grande (DSC/UFCG)[21] and the Department of Informatics e Applied Mathematics of the Federal University of Rio Grande do Norte (DIMAp/UFRN)[22].

1.6 Structure of this Dissertation

This dissertation is organized in two parts. The first part "Overview" gives a motivation for this work, the research context and background of the work done. The second part "Research Papers" contains the set of research papers produced related to this dissertation. Next, we introduce each one of the two parts of this dissertation and how they are composed.

Part I - Overview. In addition to this introductory chapter, the first part of this dissertation reviews the most important aspects relevant to understand our contributions in:

- *Chapter 2 - Background* provides a review of the base concepts, techniques, tools and practices related to the subject of this dissertation, such as Software Product Line Engineering, Model-Driven Development and consistency checking. This chapter is not intended to provide an historical overview nor a comprehensive survey. It rather concentrates on the aspects we think are more useful to understand this dissertation and to highlight the relationships between them.

- *Chapter 3 - DCC4SPL Approach* gives an overview of our approach and highlights the relationships between its components, VML4RE and VCC, as well as their respective tool support.

- *Chapter 4 - Validation* addresses the questions raised by each research topic and the methods used to answer them.

- *Chapter 5 - Conclusions* summarizes the main contributions of this dissertation and identifies future work.

Part II - Research Papers contains a copy of some of the main book chapters and peer-reviewed papers accepted and presented in international events describing the main results of this dissertation.

[19] http://www.sea.uni-linz.ac.at/
[20] http://www.cin.ufpe.br/
[21] http://www.dsc.ufcg.edu.br/~spg
[22] http://www.dimap.ufrn.br/

- *Chapter 8 - A Model-Driven Approach for Software Product Lines Requirements Engineering* [9] presents how model-driven techniques can be used to automatically derive requirements models for specific products of an SPL, and traceability views that explicitly illustrate the relationships between features and UML requirements model elements.

- *Chapter 9 - Multi-View Composition Language for Software Product Line Requirements* [15] presents details about the design criteria and use of the Variability Modeling Language for Requirements (VML4RE). This chapter and the previous one set the bases for the VML4RE description provided in Chapter 3.

- *Chapter 10 - VML* – A Family of Languages for Variability Management in Software Product Lines* [97] presents how to build new VML* languages for new SPL contexts avoiding error-proneness of language development. Some features of VML* such as the use of feature expressions instead of only atomic features were exported to new versions of VML4RE that improved the language implementation described in paper [15].

- *Chapter 11 - Model-Driven Requirements Specification for Software Product Lines* [13] presents different approaches for specifying and composing requirements in SPLs as well the motivation and description of some of the design patterns followed by DCC4SPL.

- *Chapter 12 - Evaluating Approaches for Specifying Software Product Line Use Scenarios* [6] presents an empirical evaluation of four key and representative techniques that aim at a better management of common and variable use scenario specifications between products of an SPL. This evaluation takes into account relevant quality attributes for variability management, such as modularity, expressivity and stability. This evaluation is one of the key parts of Chapter 4 - Validation.

- *Chapter 13 - Supporting Consistency Checking between Features and Software Product Line Use Scenarios* [12] presents an approach whose driving objective is to enable consistency checking in the problem space between requirements models and features.

- *Chapter 14 - Ensuring Consistency Between Feature Models and Model-Based Specications - The VCC Approach* [11] proposes an approach to support consistency checking between a feature model and its corresponding model-based specifications. The resulting approach is called Variability Consistency Checking (VCC). Also, this paper exemplifies how to use VML4RE composition language with VCC. This paper supports the consistency checking activity of DCC4SPL provided in Chapter 3.

There are also more authored publications that inspired and supported the work presented in this dissertation, e.g., [49, 88, 96, 78, 2, 3, 8, 33, 7, 60, 10, 5, 4].

2

Background

This chapter lays the foundations to understand the subjects related to this dissertation. It is composed of three sections: "Overview" offers a glimpse of the fundamentals, approaches and models employed, "Fundamentals" presents some concepts in software engineering, and "Approaches and Models" provides an overview of some approaches and their corresponding models where those fundamentals are applied.

2.1 Overview

This section introduces the essential concepts and their relationships required to understand this dissertation. To ease the understanding of the whole "puzzle" of concepts, Figure 2.1 shows an informal sketch representing some of the main fundamentals, approaches, models, and their relationships presented in this dissertation. The most important concepts introduced here will be further explained in Sections 2.2 and 2.3.

Figure 2.1 shows that *Software Product Line Engineering (SPLE)* applies planned *reuse* of software assets in a domain. In SPLE, a functionality or non-functional property is called *feature* and each feature can be common to all (or a subset of) the products in a domain or be unique (i.e., varies) for some products. SPLE manages *variability* and *commonality* by supporting the overall process to identify common and variable features, as well as to manage the set of products configurations. During *feature modelling* developers create a *feature model* that presents the available features in an SPL and shows the constraints on their usage. For example, the selection of one feature may preclude or require the selection of other features.

To create a new product developers perform a configuration process that consists of a selection of features. Next, developers compose the assets related to the selected

features with the assets related to common features. The result of this composition is the *customization* of reusable assets in the context of a specific software product variant. There is not standard way to customize reusable assets for specific product variants. In our approach, the way to describe how to weave the assets related to variable features with those related to common features is through a *composition specification*.

In what concerns supporting technologies for SPLE, our approach uses *Model-Driven Development (MDD)* and *Domain-Specific Language Engineering (DSLE)*. MDD is driven by *modelling* and resulting *models* that represent different perspectives of a system. In MDD the act of writing models using a language is called *modelling*, while the act of writing a language (i.e., modelling a language) is called *metamodelling*. Models can be transformed to other model(s) usually to produce a translated model written in other language or to customize some of its *model fragments* (i.e., to perform a *model transformation*). There are languages such as UML that can be used to model requirements and architecture. Also, it is possible to create custom-made languages that employ a vocabulary that is specific to a domain using DSLE. DSLE and MDD are related very closely. DSLE is supported by MDD which provides the conceptual bases for metamodelling, modelling and using the resulting models as the main development assets. MDD is implemented using DSLE which provides specialized languages for writing models, metamodels and model transformations.

Principles such as *Separation of Concerns (SoC)* aim at improving modularization of systems. SoC advocates that developers should decompose a system in such a way that each one of the resulting decompositions (e.g., models, classes, packages) implements, if possible, only one concern. If each part of a system is focused on a special concern of the system, each part can be modified easily and with fewer side effects on other parts. Applying SoC in SPL modelling eases to reason about *software composition and decomposition* encouraging developers to implement and evolve concerns (e.g., a feature or set of features in an SPL) more separately, so they can be reused to compose new products.

After specifying an SPL using different models, they may evolve and changes in these models may result in inconsistencies with what is specified in the feature model. *Consistency checking* is useful in early SPLE to guarantee consistency among the various models. For example, to verify that the models that document, design and implement features do not have contradictions with respect to the specification of the feature model.

2.2 Fundamentals

This section presents the main fundamentals that form the basis for the approaches addressed in this dissertation. Next we describe briefly the list of fundamentals that compose the top part of Figure 2.1, some of them will be later extended during the description of the approaches.

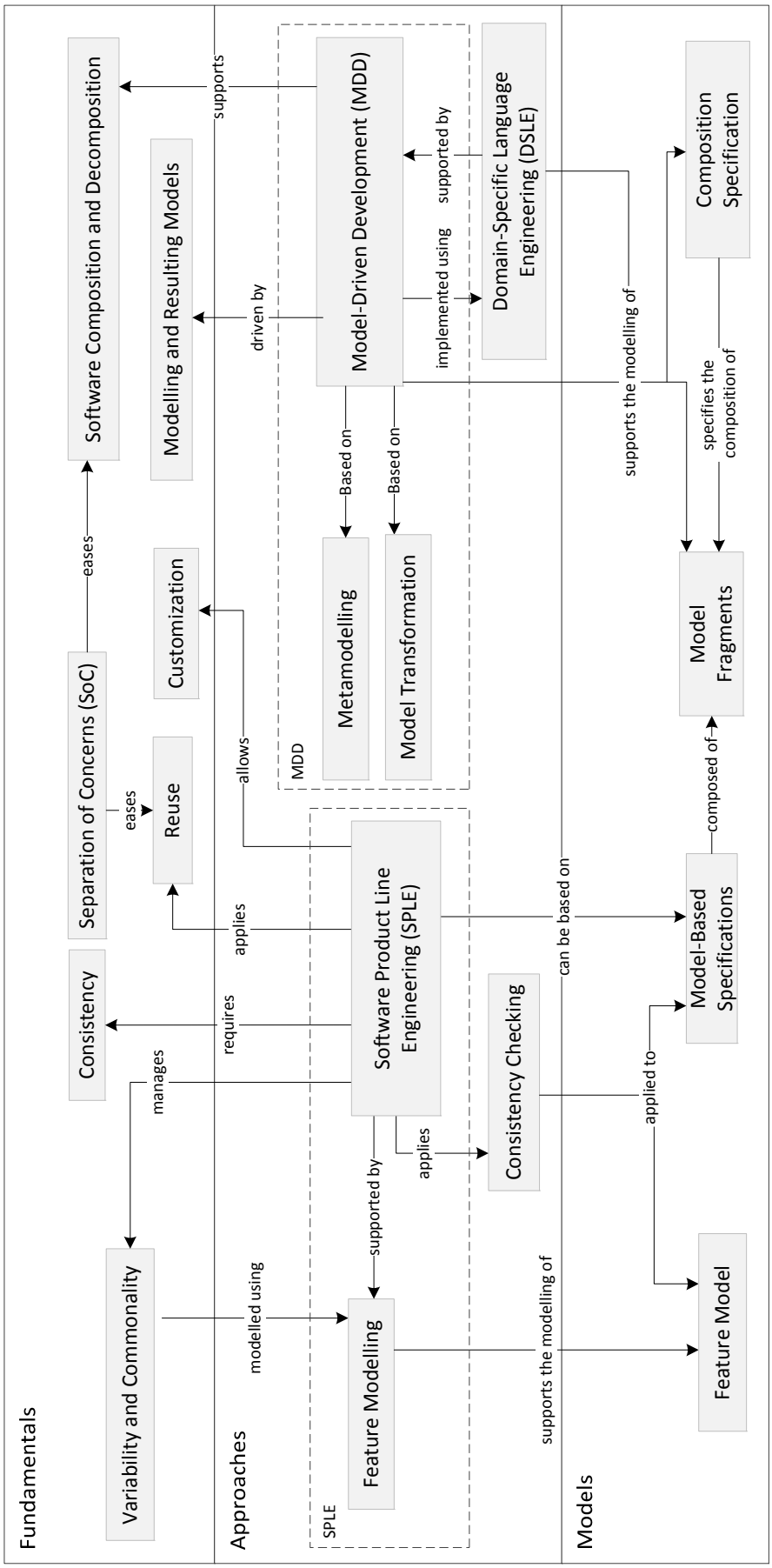


Figure 2.1: Informal sketch representing some of the fundamental concepts, approaches, models, and their relationships presented in this dissertation.

2.2.1 Separation of Concerns (SoC)

This is a fundamental principle of software engineering that is credited to Dijkstra [36, 35] and Parnas [76, 77]. It states that in software development it is easier to manage a problem by breaking it down into smaller pieces than to solve the problem as is. Such pieces are the *concerns* of a software system, where a concern is a semantically coherent issue of interest to the problem domain. Concerns are the primary criteria for decomposing software into smaller, more manageable and comprehensible parts. Thus, we should decompose a program in such a way that each one of the resulting modules implements one single concern. Indeed, a non-modular design, without a clear separation of concerns leads to bad modularization (resulting in scattering and tangling of concerns) that advanced software engineering approaches, such as Aspect-Oriented Software Development (AOSD), aim to address [38, 19, 44].

2.2.2 Software Composition and Decomposition

In all phases of the software life cycle, concerns of a software system are separate pieces, distinguishable from other concerns. However, such separation is non-trivial to achieve, especially in large-scale and evolvable software [16]. Design and implementation techniques have to support separation of concerns explicitly by providing appropriate (de)composition mechanisms. Decomposition means to break down a software design into pieces; composition ties these pieces together to get a complete software product. Design and implementation techniques have to provide different kinds of (de)composition mechanisms at different levels of abstraction in order to account for the diversity of possible concerns. Prominent examples of decomposition modules are the concepts of functions in structured programming and classes in Object Oriented Programming (OOP). While functions decompose a software system along its instructions, classes decompose a software system along the data to be encapsulated [16].

2.2.3 Modelling and Resulting Models

"Modelling, in the broadest sense, is the cost-effective use of something in place of something else for some cognitive purpose. It allows us to use something that is simpler, safer, or cheaper than reality" [80]. A model represents a perspective of reality for a given purpose; the model is an abstraction of reality as it cannot represent all aspects of reality. This allows us to deal with the world in a simplified manner, avoiding the complexity, danger and irreversibility of reality" [80]. We complement this definition of model with another specific to software development that states that "a model is a description of (part of) a system written in a well-defined language" [59], where a well-defined language is "a language with well-defined form (syntax), and meaning (semantics), which is suitable for interpretation by a computer" [59].

2.2.4 Consistency

A system is consistent when there is logical coherence (i.e., no contradictions) bewteen its parts. Consistency depends on the verification of consistency conditions that should be guaranteed in a system. These can be as diverse as architectural and design guidelines, programming conventions and well-formedness rules. Consistency conditions may involve several different and interrelated models required to represent the perspectives (also called: *views* or concerns) and information needed by diverse system stakeholders [40]. A traditional example of consistency conditions (or consistency rule) among different models is that "if a sequence diagram has a message M whose target is an object of class C, the class diagram of class C must contain method M" [66].

2.2.5 Reuse

Software reuse is the process of creating software systems from existing software rather than building software systems from scratch [63]. Separated concerns in software development can be more easily reused in different contexts than intermingled ones. The more independent a concern is, the easier it can be detached from or attached to a (different) software system. Reuse can be applied to employ part of the implementation or design of a concern (e.g., a use scenario, a component, a model, or library) in different variants of one software product or in different software systems.

2.2.6 Customization

Customizing a software system means adjusting the given system structure and behaviour in the boundaries of the supported variability [93]. This means to choose the concerns desired for a system and to select those implementations that best fit a requirements specification. Customization is required to meet the specific needs of different stakeholders on a software system, therefore, software design and implementation ideally should support the easy customization and derivation of system variants.

2.2.7 Variability and Commonality

Software variability represents the need of a software system or artefact to be changed, customized or configured for use in different contexts [93]. High variability means that the software is more reusable and, therefore, it may be used in a broader range of contexts. The degree of variability of a particular system is given by its variation points, which are the parts that support configuration and consequently customization of the product for different contexts or for different purposes. The identification of what is subject to change is intimately related to that of what does not change, i.e., commonality. Variability and commonality are base concepts in SPLE, which will be discussed further in Section 2.3.1.

2.3 Approaches and Models

This section introduces the approaches that will be addressed in this dissertation and the models that they use or help to produce. The middle part of Figure 2.1 shows four main approaches: Software Product Line Engineering (SPLE), Model-Driven Development (MDD), Domain-Specific Language Engineering (DSLE) and Consistency Checking. They are related to other sub-approaches shown into dashed lines: feature modelling, metamodelling and model transformation. All approaches are related to the fundamentals explained in Section 2.2 and use or help to produce different kinds of models (shown at the bottom of Figure 2.1).

Software Product Line Engineering (SPLE) is the application domain of this dissertation. SPLE is supported by feature modelling which is a key technique employed to describe variability and commonality in an SPL and to produce a feature model. This dissertation employs the feature model as one of the input artefacts in SPLE.

Model-Driven Development (MDD) is described through two of its distinctive techniques: metamodelling and model transformations. A connection to Domain-Specific Languages Engineering (DSLE) is established where MDD techniques, such as metamodelling, are applied to create new domain-specific languages. This dissertation uses DSLs to create languages to express composition specifications and the actual transformation of the models.

Consistency Checking is an activity applied to guarantee logical coherence between different interrelated parts in a system. Some common types of consistency checking are described in this section. This dissertation applies consistency checking to guarantee that the models employed to specify an SPL, such as feature model and model-based specifications, are logically coherent. Consistent models are more reliable as they have not contradictions between them, therefore, they can be used to derive product-specific models.

2.3.1 Software Product Line Engineering (SPLE)

Software Product Line Engineering is an approach to increase software quality and productivity through systematic reuse of assets [79, 27, 31]. SPLE encompasses the creation and management of products' families for a particular domain. In SPLE each product in the family is derived from a shared set of *core assets*, following a set of prescribed rules [27]. A core asset can be any relevant artefact in the software process, such as code units (packages, classes, methods, etc.) or models, documentation, configuration files, etc.

Products in an SPL are characterized by their *features*, which are useful to express product functionalities or properties concisely [79]. Features may be common to all products or vary between products. The terms *commonality* and *variability* are often used to denote the common and variable features within a product line, respectively.

To facilitate the understanding of our proposal we briefly present the Pohl et al. SPLE

process conceptual framework [79] as well as an overview on feature modelling. We chose this conceptual framework because it considers that all kind of artefacts may be reused, from requirements through code, to testing units. However, in comparison with our work, it does not address the development of effective solutions for models derivation and consistency checking based on MDD and DSLE.

SPL Process Framework [79]. This conceptual framework uses a collection of *reusable artefacts* as input and offers the possibility of mass customization. *Reusable artefacts*, or *core assets*, encompass not only code-based units such as packages, libraries and components, but also all types of software development artefacts such as requirements models, architectural models, test plans, and test designs. According to Pohl et al., "to facilitate mass customization, the platform (i.e., the collection of reusable artefacts[1]) should provide the means to satisfy different stakeholder requirements. For this purpose, the concept of variability is introduced in the platform. As a consequence of applying this concept, the artefacts that can differ in the applications of the product line are modelled using variability" [79].

Figure 2.2 shows an adaptation of the original figure of the Pohl's SPLE conceptual framework [79][2]. It has two main processes: *Domain Engineering* (also known as *Core Asset Development*) and *Application Engineering* (also known as *Product Development*) [79, 27].

Domain Engineering is the process responsible for "establishing the reusable platform and thus for defining the commonality and the variability of the product line" [79]. Traceability links between these artefacts facilitate systematic and consistent reuse. Product management is also part of the domain engineering process and deals with the economic aspects of the SPL such as the market strategy and the management of the product portfolio of the company. Domain engineering sub-processes include:

- *Domain Requirements Engineering* encompasses all activities for eliciting and documenting the common and variable requirements of the product line. The output of this sub-process comprises the common and variable requirements for all foreseeable applications[3] of the product line as well as a variability model (e.g., a feature model)[4].

- *Domain Design* encompasses all activities for defining the reference architecture of the product line. The reference architecture provides a common, high-level structure for all the product line applications. The input for this sub-process consists of the domain

[1]The definition of platform can be also extended to the technologies on which other technologies or processes are built.

[2]Our adaptation considers the cases where: (1) variabilities and commonalities may be modelled in the same model. This is represented by single geometrical shapes with white and black parts representing models with fragments related to commonalities and variabilities, respectively; and (2) models used in each sub-process of SPLE can be modelled separately; this is represented by disconnected shapes.

[3]Along this dissertation we also refer to "applications" as the "products" or "product-variants" of the SPL.

[4]In Figure 2.2 sub-processes such as domain requirements engineering should include different kinds of geometrical shapes representing the different kinds of models created by each stakeholder. However, we only use one kind of geometrical shape by sub-process to avoid adding unnecessary extra complexity in the figure.

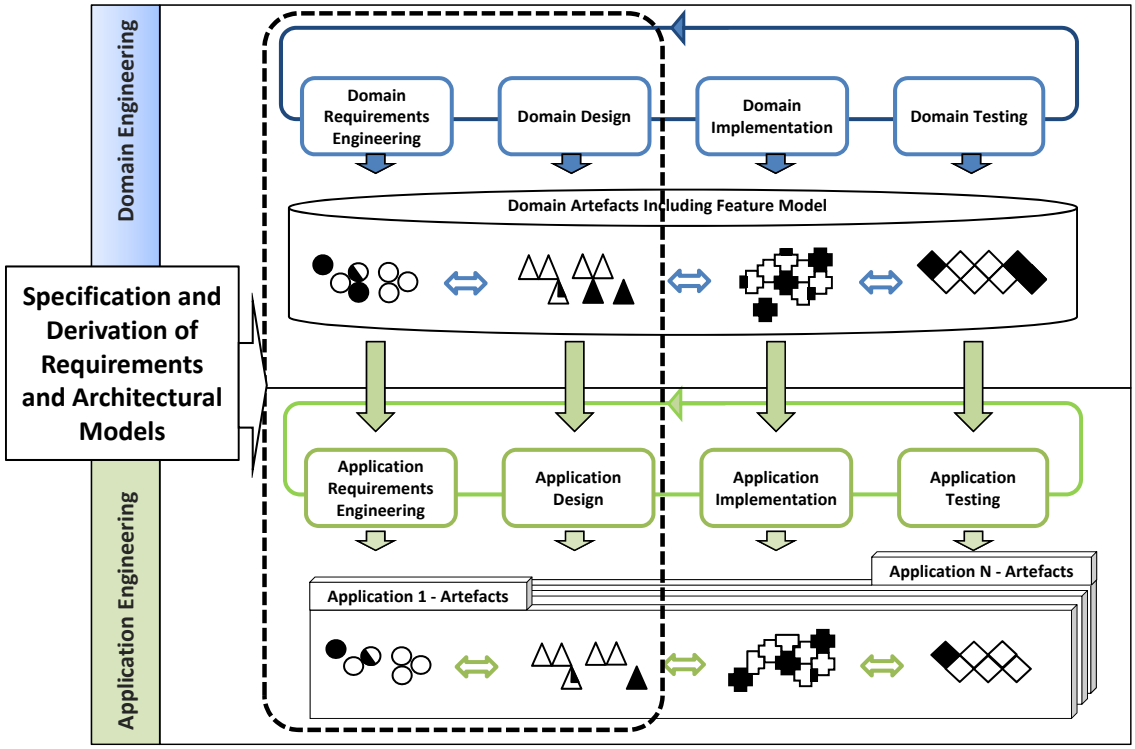


Figure 2.2: SPL process framework (adapted from [79]).

requirements and the variability model from domain requirements engineering. The output encompasses the reference architecture and a refined variability model that includes the so-called *internal variability*, i.e., the variability that is necessary for technical reasons.

- *Domain Realization* deals with the detailed design and the implementation of reusable software components. The input of this sub-process consists on the reference architecture including a list of reusable software artefacts to be developed in domain realization. The output of domain realization encompasses the detailed design and implementation assets of reusable components.

- *Domain Testing* deals with the validation and verification of reusable components. Domain testing tests the components against their specifications, i.e., requirements, architecture and design artefacts. In addition, domain testing develops reusable test artefacts to reduce the effort for application testing.

Application Engineering is the process responsible for "deriving product line applications from the platform established in domain engineering" [79]. Application Engineering exploits the commonality of the product line and ensures the correct binding of the variability according to the applications' specific needs. The application engineering sub-processes are:

- *Application Requirements Engineering* encompasses all activities for developing the requirements specifications for specific applications.

- *Application Domain Design* encompasses all activities for producing the application architecture.
- *Application Realization* encompasses all activities for creating a running application together with the detailed design artefacts.
- *Application Testing* encompasses all activities necessary to validate and verify an application against its requirements.

It is important to note that neither the domain and application engineering nor their sub-activities have to be performed sequentially. This is expressed in Figure 2.2 with a loop with an arrow for each process. A dotted line marks the part of the framework that is directly related with the subject of this dissertation. We also make explicit that there is a composition process that derives models for specific target products (or "applications" in Figure 2.2) from reusable requirements and architectural models for the entire SPL created during *Domain Requirements Engineering* and *Domain Design*.

Another important aspect of this framework is that testing does not consider consistency between the variability model and other assets, focusing only on testing implemented components against their requirements and architecture specifications. In this dissertation we address some of the gaps in the SPL process considering consistency between feature models and other model-based assets (e.g., requirements and architecture models) and automatic derivation of model-based assets.

Feature Modelling. Feature modelling is a method and notation to capture the available common and variable features of the systems in a family and to describe the incompatibilities and dependencies between features [53, 31].

Feature modelling was first proposed by Kang et al [53] and since then it has been extended with several concepts. For example, feature and group cardinalities, attributes, and diagram references [32]. A classic example of a feature model is illustrated in Figure 2.3. This feature model expresses an electronic commerce system that supports one or more different payment methods, provides tax calculation taking into account either the street-level address, or postal code, or just the country, and it may or may not support shipment of physical goods. Such feature model may be complemented with additional information, such as binding times (features may be intended to be selected at certain points in time), domain constraints (e.g., dependencies and incompatibilities, where selecting a certain feature may require or exclude the selection of another feature), default attribute values and default features, stakeholders interested in a given feature and priorities [32].

DeBaud and Shmith show that feature models are useful in the early stages of software family development [34]. They provide the basis for scoping an SPL by recording and assessing information such as which features are important to enter a new market or remain in an existing market, which features incur a technological risk, what is the

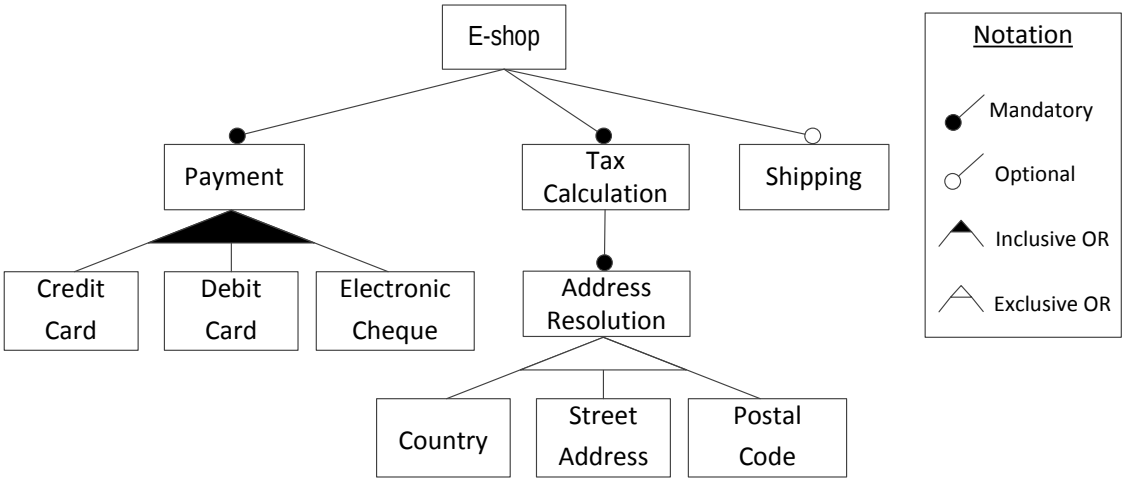


Figure 2.3: Example of feature model for an E-Shop SPL [32].

projected development cost for each feature, and so forth. In the context of generative software development [31], feature models are used during domain analysis are the starting point in the development of both SPL architecture and DSLs.

2.3.2 Model-Driven Development (MDD)

Model-Driven Development (MDD) refers to the systematic use of models as first-class entities throughout the entire software lifecycle [85, 59]. MDD is based on metamodelling, which helps to express well-defined languages, and model transformations, which automate the software development process. This section introduces these base MDD techniques and presents Domain-Specific Languages (DSL) as a way to put MDD into practise.

Metamodelling. Metamodelling can be defined as the analysis, construction and development of the frames, rules, constraints, models and theories applicable and useful for modelling a predefined class of problems [94]. In other words, metamodelling is the practice of constructing models that specify other models.

A metamodel is known as the abstract syntax of a language and encompasses all the constructs that can be used in that language. A model is said to conform to its metamodel, or that it is an instance of its metamodel in the way that a particular computer program conforms to the grammar of the programming language in which it is written.

The left-hand side of Figure 2.4 presents a four layers metamodelling framework. This framework is exemplified using MOF, a modelling formalism employed to define metamodels [5]. The right-hand side of Figure 2.4 depicts a short example based on the modelling of a film released as DVD. It can be interpreted as follows:

- *M0, Data Layer* shown in the left hand side of Figure 2.4 describes the instances

[5]http://www.omg.org/mof/ : OMG's MetaObject Facility (MOF)

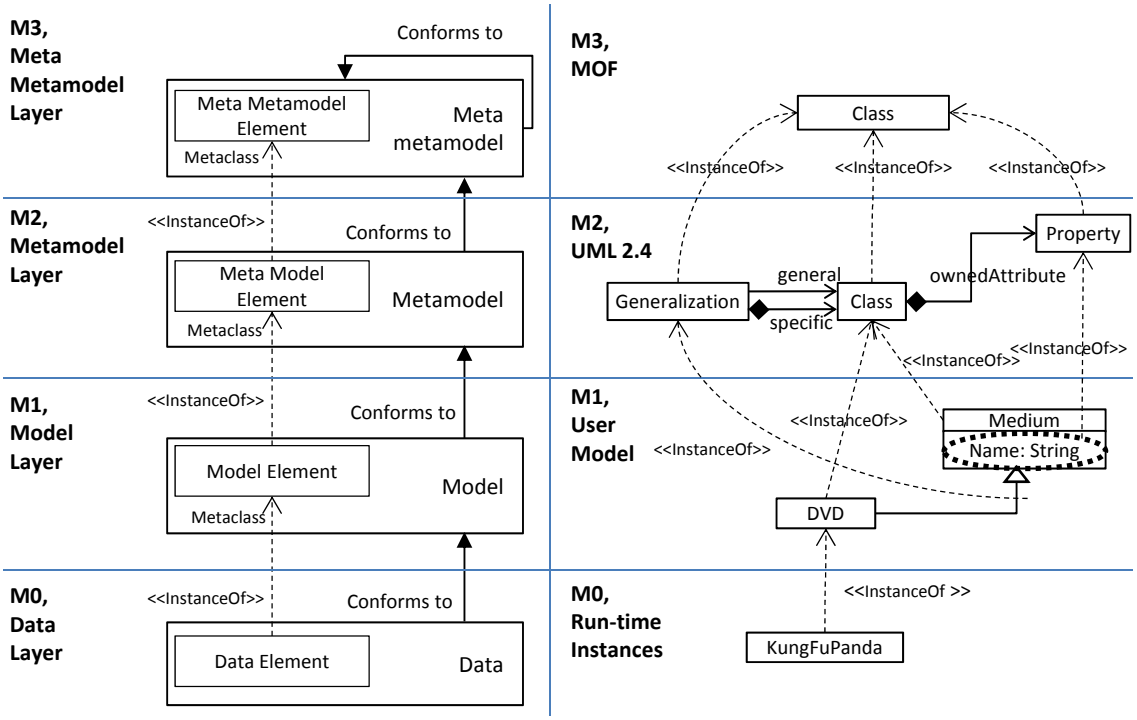


Figure 2.4: (Left) Four-layers metamodel hierarchy, (Right) An example of the four-layer metamodel hierarchy using Meta-Object Facility (MOF).

of the models in layer M1. Usually M0 contains data, runtime instances of user created objects, or source code derived from the M1 layer for different programming languages (e.g. Java and C++). In MOF, M0 contains the instances, sometimes referred to as *runtime* instances, of model elements defined in a model contained in the M1 layer. The corresponding right-hand side level represents an instance, "Kung Fu Panda", of the model element DVD.

- *M1, Model Layer* describes the information in layer M0. M1 contains the models from which the data, objects, or source code in M0 is generated. The models written using UML, such as the class diagram in Figure 2.4 (right-hand side) are examples of models contained in the M1 layer.

- *M2, Metamodel Layer* describes the models in layer M1. M2 contains the metamodels that define the abstract syntax of the languages needed to write model elements in M1. Metamodels determine which kind of elements can be present in the model at the M1 level. In MOF, M2 is represented by UML which contains metamodel elements that are the metaclases of the model elements in M1 (e.g., "Class", "Attribute", and "Generalization").

- *M3, Meta Metamodel Layer* describes the metamodels in layers M2 and M3. Similarly to EBNF, MOF could be defined in MOF; this is represented in Figure 2.4 as the recursive relationship of "conforms to" in the M3 layer.

An example of a MOF implementation is EMF [85], which is an open source framework targeted to Java. EMF defines Ecore as meta metamodel and can create Java code for graphically editing, manipulating and serialising data based on a model specified in an XML Schema, UML class diagrams, or annotated Java classes. The tool support Section 3.6 of this dissertation will elaborate on how we employ this technology.

Model Transformations. Model transformations can be seen as taking a set of input parameters including source and target models, and producing a set of target model elements as output [83]. Consequently, there is a mapping relation between the source model elements and the target model elements (a one-to-many or many-to-one relationship).

There are two main approaches for models transformation:

- *Graph transformation systems*. These use graph rewriting techniques to manipulate graphs [81]. A graph transformation is defined in terms of a set of production rules. A production rule consists of a Left-Hand Side (LHS) graph and a Right-Hand Side (RHS) graph. Such rules are the graph equivalent of term rewriting rules; intuitively, if the LHS graph is matched in the source graph, it is replaced by the RHS graph.

- *Program transformation techniques*. These are also known as *Meta-Programing* or *Generative Programing* [31]. They are used in many areas of software engineering, ranging from program synthesis to program optimisation and program re-factoring, to reverse engineering and documentation generation. Program and model transformation is an approach that allows automatic generation of software from a generative domain model [31]. A generative domain model is a model of a system family that consists of a problem space, a solution space, and the configuration knowledge. The problem space defines the appropriate domain-specific concepts and features. The solution space defines the target model elements that can be generated and all possible variations [72]. The configuration knowledge specifies illegal feature combinations, default settings, default dependencies, construction rules, and optimisation rules. Generative Programming introduces generators as the mechanisms for producing the target.

2.3.3 Domain-Specific Language Engineering (DSLE)

Some of the MDD approaches are based on Domain-Specific Language Engineering (DSLE). DSLE provides the technology and methodology to design and implement DSLs. A DSL offers expressive power for a particular problem domain, such as a specific class of applications or application aspect [18]. DSLs can be implemented in different ways: textual or graphical languages, interactive GUIs (wizards, editors, forms), or as extensions of other programming languages [46]. Different tools and platforms are now being defined

to support DSL implementation and processing, such as, Microsoft Visual Studio Visualization and Modeling SDK[6], Generic Modeling Environment (GME)[7], Eclipse Modeling Project [8] and MetaEdit+ [9].

Advantages over General-Purpose Languages. In comparison with general purpose programming languages, such as Java or C#, that were designed to be appropriate for any kind of application domain, DSLs simplify the development of applications in specialized domains at the cost of their generality. A DSL can offer several important advantages over a general purpose language [29]:

- *Domain-specific abstractions*. A DSL provides pre-defined abstractions to represent concepts directly from the application domain.
- *Domain-specific concrete syntax*. A DSL offers a natural notation for a given domain and avoids syntactic clutter that often results when using a general-purpose language.
- *Domain-specific error checking*. A DSL enables building static analysers that can find more errors than similar analysers for a general-purpose language and that can report the errors in a language familiar to the domain expert.
- *Domain-specific optimizations*. A DSL creates opportunities for generating optimized code based on domain-specific knowledge, which is usually not available to a compiler for a general purpose language.
- *Domain-specific tool support*. A DSL creates opportunities to improve any tooling aspect of a development environment, including editors, debuggers, version control, etc. The domain-specific knowledge that is explicitly captured by a DSL can be used to provide more intelligent tool support for developers.

An example DSL is the feature models [53, 31] used to express common and variant features among products in an SPL. In this dissertation we propose the VML4RE [15] (described in Section 3.4) which is a DSL that describes requirements models derivation.

Elements of a Language Definition. There are three main elements in a language definition which are generic to textual languages as well as graphical languages [58]:

- *Abstract syntax* defines the concepts of a language and their relationships. It is independent of any particular concrete syntax representation of the language. Metamodelling is a popular method to define the abstract syntax of languages. It simplifies the language development by allowing developers to map classes of a domain analysis model to metaclasses in a metamodel.

[6] http://code.msdn.microsoft.com/vsvmsdk
[7] http://www.isis.vanderbilt.edu/Projects/gme/
[8] http://www.eclipse.org/modeling/
[9] http://www.metacase.com

- *Syntactic mapping* consists of a set of rules that relates the concepts of a language (in the abstract syntax) to their representation (in a concrete syntax). The concrete syntax of a language may be textual or graphical. For textual languages the concrete syntax defines how to form sentences while for graphical languages defines the graphical appearance of the language concepts and how they may be combined.

- *Semantics* describes the meaning of a program or model specified in some language. Often, language definers explain the meaning of concepts in the abstract syntax informally through examples and plain natural language [47]. However, various ways have been developed to describe the semantics of languages in more rigorous ways [58]. A popular way to express semantics is *operational semantics*, where the meaning of a language construct is specified by the computation it induces when it is executed on a machine. Concepts in the abstract syntax relate to semantic domain elements (e.g., which varies from plain natural language to rigorous mathematics) through a *semantic mapping*.

2.3.4 Consistency Checking

Using multiple views to describe software, each one focusing on a perspective of the system (e.g., behaviour, structure, or variability), allows to a more exhaustive modelling of the system [61, 73]. Using specialised views reduces the complexity of one single view, making it easier for a developer to build correct models [61, 73]. However, modelling software using multiple views implies the need for consistency between them.

Some research has been done in classifying and dealing with consistency problems related to multi-view modelling, for example [91, 67, 39, 66]. The five most relevant classes of consistency identified are:

- *Intra-model consistency,* applied to exactly one model type (e.g., to check name crashes or unconnected items [39]).

- *Inter-model consistency,* applied to multiple model types (e.g., to check that the class names used in a sequence diagram appear in their associated class diagram [67]).

- *Syntactic consistency*, applied to check that a model conforms to its language definition, usually specified by its metamodel and language grammar (e.g., to check that a model does not contain any model element that is not defined in its language).

- *Semantic consistency,* applied to check that the semantic information a modeller is able to derive from different models must be compatible, i.e., not contradictory (e.g., to check that the events produced in a sequence diagram should not produce inconsistent states in the state diagrams of the objects that participate in the interaction).

In this dissertation we focus on inter-model and semantic consistency. The other types of consistency (intra-model and syntactic) have been studied intensively in SPLE. Intra-model consistency in SPLE is mainly focused on feature model analysis and it has been comprehensively documented by other authors (e.g., [23, 21, 22]). Syntactic consistency is addressed for example by EMF that allows generating editors from metamodels. The models created with these editors allow creating only models that conform to their metamodels.

Consistency checking in SPLE is a complex task given that the number of feature combinations and dependencies between model elements can grow exponentially [57, 90]. To date, there is no approach that guarantees that the feature model is semantically consistent with the requirements and architectural models that design its features. Our approach helps to fill this gap with an automated technique to detect inter-model semantic inconsistencies between feature model and other models created during domain engineering, showing the model elements involved in each inconsistency and the cause of the inconsistency.

2.4 Summary

The application domain of this dissertation is Software Product Line Engineering. This domain is located in the broader area of strategic, planed reuse of software artefacts, from which we focus on models specification, verification and derivation. This chapter summarizes and associates the fundamental concepts, approaches and models applied in our work.

Model-Driven Development employs models as the primary development asset. This means that models are the input of the product derivation process that is implemented as transformation of models. Models are written using modelling languages such as UML or any specific DSL. These modelling languages are defined using metamodels that define and restrict the abstractions that can be used during the models specification process.

Another important concept in this chapter is "feature modelling". Feature modelling is a method and notation to capture the available common and variable features (i.e., functionalities and qualities) of the products in an SPL and describe the incompatibilities and dependencies between the features.

Given that feature models and other requirements and architectural models are created and evolved separately they may become inconsistent. Hence, consistency checking is necessary to verify that there are no violations of consistency conditions that ensure that what is written in the feature model does not contradict what is represented in other models and vice-versa. To ease the consistency checking process, it is necessary to process the models by software tools that can understand the language in which the models are written.

The next chapter will present our approach to models derivation and consistency checking in SPLE.

3

DCC4SPL Approach

Derivation and Consistency Checking of models in early SPLE (DCC4SPL) integrates the solutions that contribute to tackle our research question "*How to guarantee effective consistency checking and derivation of product-specific models in early Software Product Line Engineering?*". DCC4SPL is based on Model-Driven Development (MDD) and Domain-Specific Language Engineering (DSLE). MDD is used to perform the derivation of product-specific models through models transformations, and to ease the integration of consistency checking based on the interpretation of the models and the languages used to write them. DSLE is used to create languages to help developers to abstract from much of the implementation details of writing model transformations used to derive product-specific models.

This chapter first provides an overview of DCC4SPL and next presents its two main parts, the Variability Modelling Language for Requirements (VML4RE) and the Variability Consistency Checker (VCC). Finally, this chapter describes the tool support developed for DCC4SPL.

3.1 DCC4SPL Process Overview

DCC4SPL is organized as a set of activities that involve core asset development (Domain Engineering) and product development (Application Engineering) using the core assets. Figure 3.1 shows these activities and the artefacts that they produce.

Domain Engineering in DCC4SPL is achieved through repeated cycles (iterations) and in small portions at a time (incremental), allowing software developers to benefit of what was learned during earlier iterations. In DCC4SPL there are new iterations when the Consistency Checking Report generated by VCC shows inconsistencies between

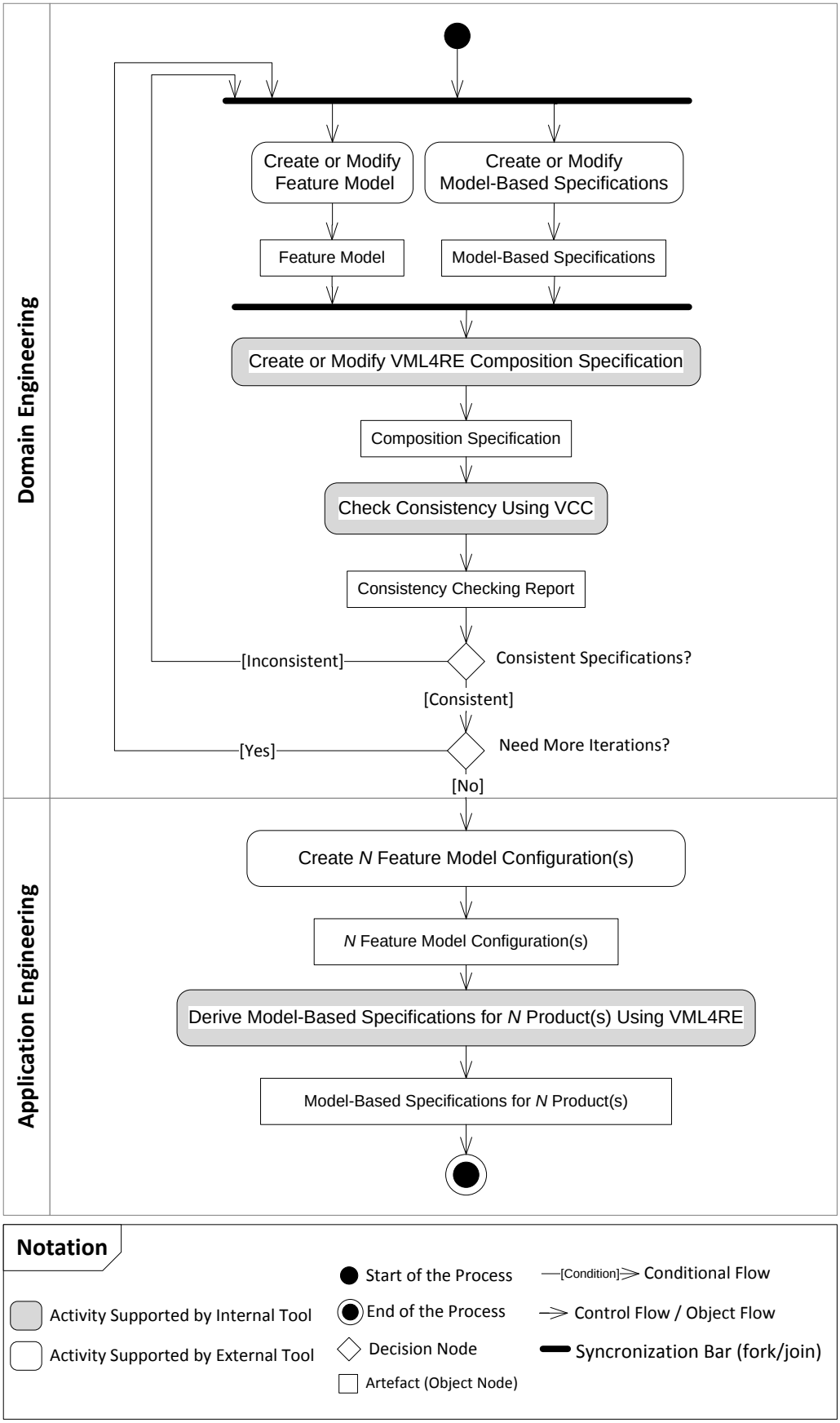


Figure 3.1: DCC4SPL main activities and artefacts.

the Feature Model and the Model-Based Specifications. If there are no inconsistencies, developers may consider either to modify the models to address new requirements or proceed to Application Engineering. Next, we summarize the Domain Engineering activities in Figure 3.1.

- *Create or modify feature model.* A feature model captures the common and variable features as well as the incompatibilities and dependencies between features of the products in a product family, describing a set of all possible valid product configurations. There are different notations to specify feature models and most of them can be translated to propositional formulas [22]. A formula contains, for each feature, a boolean variable and expresses the constraints between features. In each formula the standard $\wedge$ (and), $\vee$ (or), $\neg$ (not), $\rightarrow$ (implies), and $\leftrightarrow$ (double implication) operators of propositional logic can be used.

- *Create or modify model-based specifications* (or *feature realizations*[1] for short). Features can be implemented or designed by model-based specifications such as requirements models, architectural models or code-based modules. Each model-based specification, in contrast to mere free-form ambiguous descriptions, conforms to a language (e.g., UML) which helps to reduce ambiguity in the specifications. In this chapter we employ use scenarios as an example of model-based specification for feature models. Use scenarios are a widely used technique that describes, step by step, how an actor intends to use a system [50]. We employ use cases and activity diagrams to describe use scenarios as they are commonly used in mainstream UML-based methods such as RUP [62].

- *Create or modify VML4RE composition specification*. Composition specifications in SPL are used to express how to derive models for specific products based on transformation of models. We use the Variability Modelling Language for Requirements (VML4RE) [97, 15] as an example of a composition specification language and derivation tool that is appropriate to customize requirements analysis models. We employ VML4RE because it offers an easy reference to model elements and their transformation. Those model transformations usually are not as complex as those typically sophisticated ones created by general purpose model transformation languages such as ATL [52], AGG [89] and QVT [75]. VML4RE is inspired in the Aspect-Orientation paradigm [44][2] and, therefore, it refers to the process of model derivation as a composition process. The composition mechanism used by VML4RE allows adding, removing, connecting, and replacing parts of models with parts of other models to finally produce the models for specific products. Section 3.4 gives more details about VML4RE.

[1] The word *realization* usually means the implementation of features in code-based modules. However, in our work we also use it to refer to the design of features using model-based specifications.

[2] http://aosd.net/

- *Check consistency using VCC.* In this activity developers employ the VCC tool to check consistency between a feature model and its corresponding model-based specifications. VCC internally employs propositional formulas to relate constraints between features (called *domain constraints* and mined from a feature model) and constraints between model elements in other models (called *model-based specifications constraints* and mined from the models that design the features). In an SPL these domain constraints and model-based specifications constraints must be consistent between them to guarantee that any valid selection of features in a feature model derives valid product-specific models. For consistency constraints that are not satisfied by the SPL, the VCC tool presents to developers the particular features and fragments in the model-based specifications involved in the violation of the constraint. Such output is useful to take informed decisions about the modifications and additions of domain constraints, model-based specifications, and its composition specification. Section 3.5 gives more details about VCC.

The models produced during Domain Engineering are used in Application Engineering to derive product-specific models. First, in Application Engineering each product is characterized by the developers by means of a specific selection of features of the Feature Model called Feature Model Configuration. In Figure 3.1 we use the letter *N* to indicate the number of feature model configurations in the SPL. Next, the VML4RE tool derives Model-Based Specifications for *N* Product(s) based on the feature model configuration(s) and the models created in Domain Engineering. We summarize the activities related to Application Engineering:

- *Create N feature model configuration(s).* While a feature model describes a set of all possible valid product configurations, a feature model configuration specifies a concrete product in terms of its features. To define a feature model configuration (also known as *product configuration* or *configuration*), developers choose optional and alternative features available in the feature model created during domain engineering.

- *Derive model-based specifications for N product(s) using VML4RE.* VML4RE derives the model-based specifications related to the feature model configurations. Given that each feature model can have more than one feature model configuration, the derivation process is performed product by product. The input for the derivation is a feature model configuration, reusable model-based specifications and a composition specification. In this process, model-based specifications (e.g., use cases and activity diagrams) are transformed to fit the specifications of particular products according to the composition specification defined during domain engineering. The choice of which transformations will be used to derive a product is based on the selection of features in its related feature model configuration and the evaluation of feature expressions contained in the composition specification.

3.2 DCC4SPL Example

To illustrate the activities and artefacts mentioned in the previous section, we have chosen a home automation SPL called Smart Home [71]. The Smart Home has a wide variety of devices and it is designed to coordinate their behaviour to fulfil complex tasks automatically. Also, it enables to its inhabitants to visualize and control some devices remotely. This system was a case study developed in the European project for Aspect-Oriented, Model-Driven Product Line Engineering (AMPLE) [3]. Due to its complexity, we will focus on a subset of features described next.

3.2.1 Create or Modify Feature Model

Figure 3.2 (a) shows part of a feature model for the *Smart Home* SPL [71]. Smart Home has four optional features, AUTOMATED WINDOWS, AUTOMATED HEATING, REMOTE HEATING CTRL and INTERNET to control the heater and other devices remotely. Also, it has a set of common features, such as MANUAL WINDOWS, MANUAL HEATING and INHOME SCREEN that will be included in all the target products produced from the Smart Home SPL. Specific product configurations can be defined selecting optional features in the feature model. Figure 3.2 (b) shows a sample product configuration of the Smart Home SPL called Product-1 that includes all the features. Figure 3.2 (c) shows another sample product configuration called Product-2 that has all features except AUTOMATED WINDOWS, and that will be used to illustrate consistency problems between features and use scenarios. Domain constraints in the feature model such as the REQUIRES relationship from REMOTE HEATING CTRL to INTERNET, can be added incrementally when other model-based specifications that design the features are created (discussed below).

3.2.2 Create or Modify Model-Based Specifications

Use cases and activity diagrams provide a description of what the products in the domain should do. Feature models determine which functionality can be selected when engineering new products from the SPL. Therefore, product requirements specifications consist of customized use cases and activity diagrams. The customization is guided by a composition specification discussed in the next subsection.

Figure 3.3 (a) (left and right-hand sides) shows two of the models that are part of the model-based specifications for Product-1. Figure 3.3 (a) (left-hand side) shows an activity diagram that depicts the possible scenarios for the use case CTRLTEMPREMOTELY that is depicted in the use case diagram of Figure 3.3 (a) (right-hand side). The activity diagram also comprises activities of use cases OPENANDCLOSEWINAUTO, ADJUSTHEATERVALUE and NOTIFYBYINTERNET. Within this activity diagram it is possible to select several scenarios that correspond to different paths. Two of all the possible scenarios are:

[3] http://www.ample-project.net .

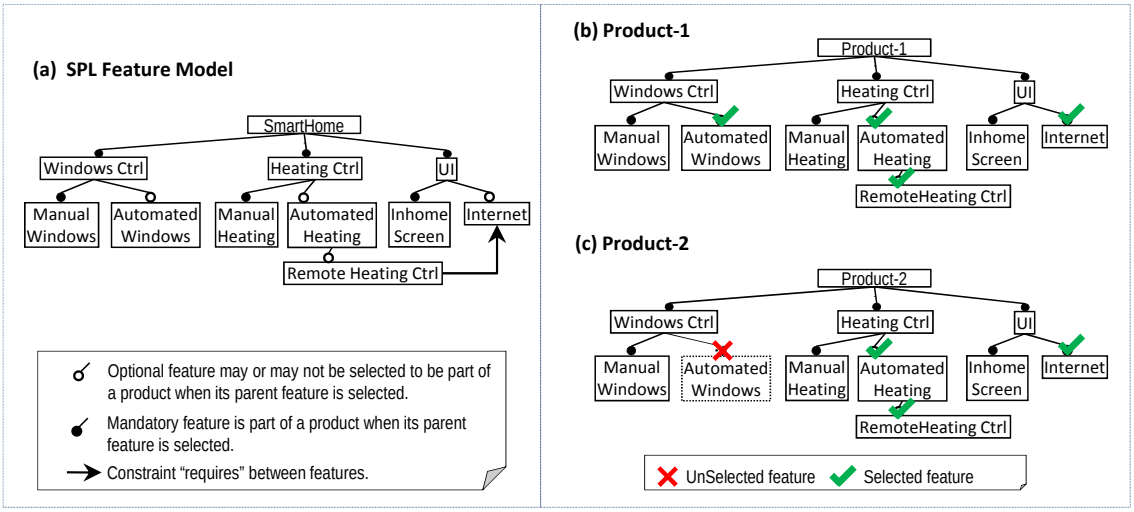


Figure 3.2: (a) Simplified sample of the Smart Home feature model, (b) Sample configuration that includes all features, and (c) Sample configuration that excludes the Automated Windows feature.

- Scenario (S1) includes reaching the in-home temperature and save energy by means of closing some windows, and
- Scenario (S2) to adjust the heater value to reach the desired in-home temperature.

Figure 3.3 (a) (right-hand side) shows part of the use case model composed for Product-1. The Include relationship (represented with «include») describes the case where one use case, the *base* use case, includes the functionality of another use case, the *inclusion* use case. The Include relationship supports the reuse of functionality in a use case diagram and is used to express that the behaviour of the *inclusion* use case is common to other use cases. Note that Include relationships between use cases may constrain the relationship between the features related to them. For example, the Include relationship between the base use case CTRLTEMPREMOTELY that includes the use case OPENANDCLOSEWINAUTO may imply that feature REMOTEHEATINGCNTRL requires the feature AUTOMATEDWINDOWS[4]. We discuss this and other constraints in Section 3.5.

The customization of model-based specifications depends on the features chosen for the SPL product, the evaluation of feature expressions (i.e., combinations of feature names and logical operators) in the composition specification (based on the selection of features), and the relationship of feature expressions with parts of the diagrams. For example, given that in Product-2 the feature AUTOMATEDWINDOWS was not selected, the variant with the feature expression "AutomatedWindows" evaluates to FALSE by the VML4RE interpreter. Therefore, the model elements related to the "AutomatedWindows" feature expression such as WINACTUATOR actor in the use case diagram as well as the swimlane (also called

[4]Developers may decide to add this implication to the feature model after VCC alerts them about it. Therefore, the version of the Smart Home feature model in Figure 3.2 (a) does not show this implication yet.

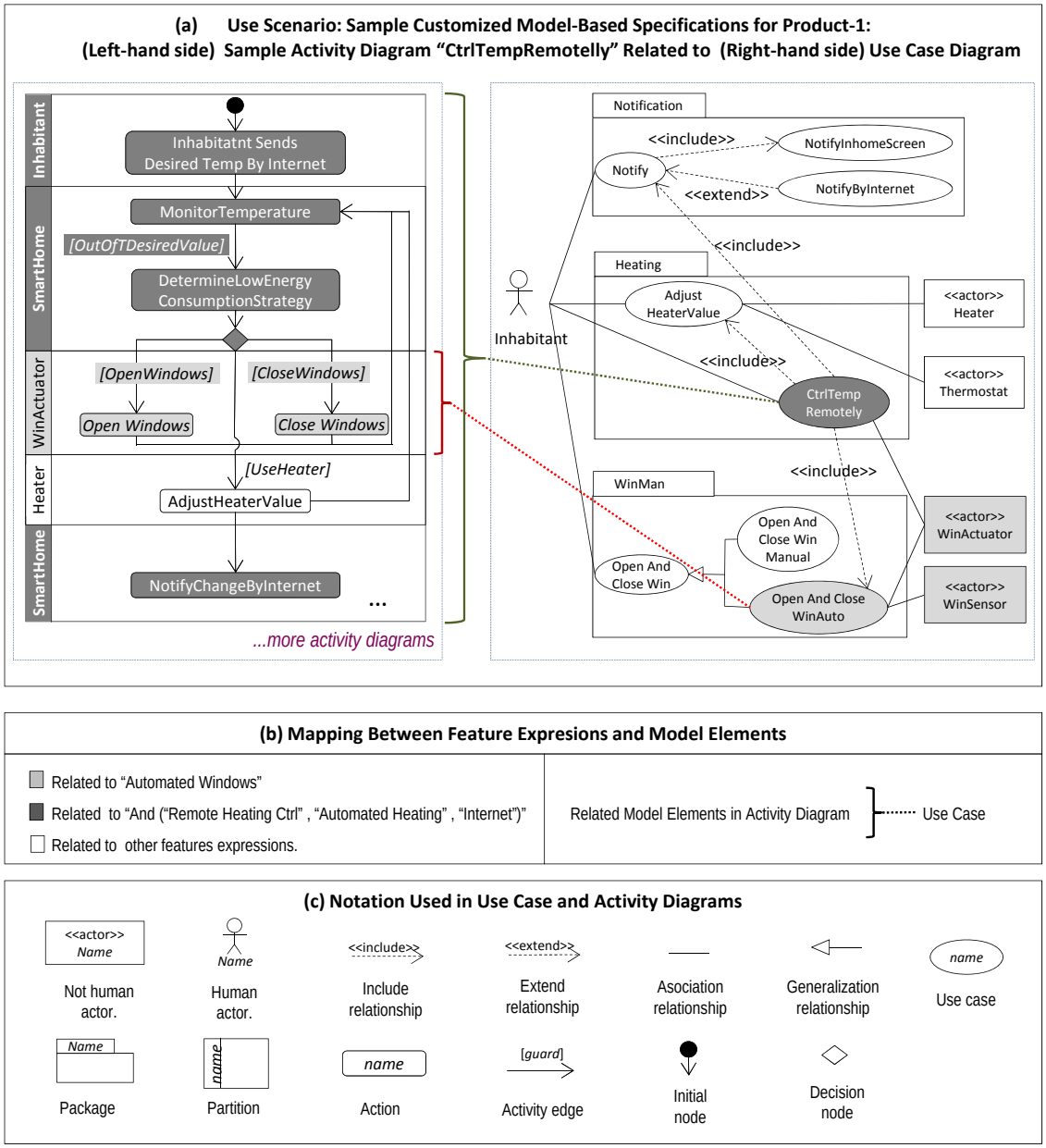


Figure 3.3: (a) Sample customized model-based specifications for Product-1, (b) Mapping between feature expressions and model fragments, and (c) Notation used in use case and activity diagram in (a).

activity partition) related to WINACTUATOR will not appear in any diagram. Therefore, scenarios such as *S1* described in a previous paragraph, are not realizable due to lack of windows actuators.

Figure 3.3 (a) presented a final composed model for a specific product that includes all the possible variable features. Therefore, we could take it as the initial model that we can transform (e.g., removing or replacing model fragments related to variable features) to obtain particular models for other products with less features. This kind of composition is called *negative composition*.

Also, we could decide to start with a small model whose parts are related to only mandatory features. Figure 3.4 shows this *core* model, that is suitable to be transformed (e.g., adding model elements related to variable features) to obtain particular models for products that include variable features. This kind of composition is called *positive composition*.

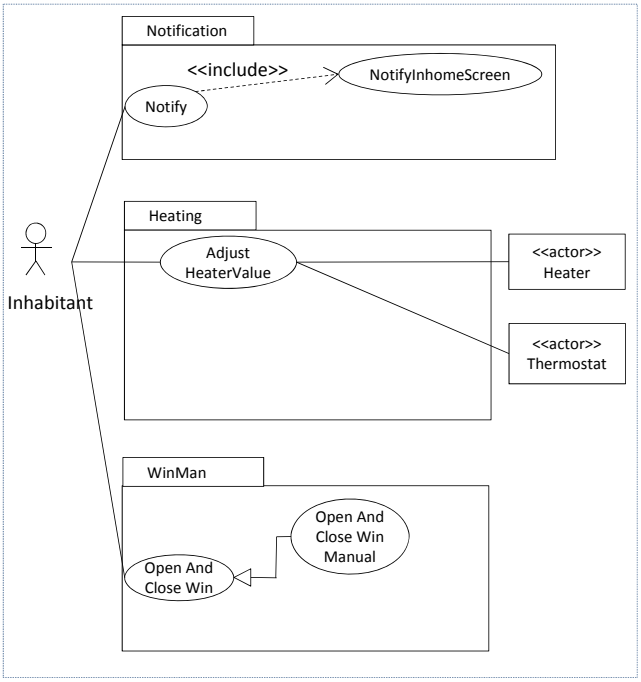


Figure 3.4: Sample use case diagram for SMART HOME containing only model elements related to mandatory features.

A combination between positive and negative variability composition is also possible, for example in cases where we want to model a core model together with some parts related to the most frequently selected variable features. Also, it may be useful to model together parts of a model that are related to dependent features. For example to model together the model fragments related to INTERNET, REMOTE HEATING CTRL and AUTOMATIC HEATING features as we did in Figure 3.3 (a).

Regardless the composition mechanism employed, it is important to employ factorization mechanisms to group common functionality. In the case of use case and activity diagrams there are inclusion and extension use cases, abstract model elements (e.g., use cases and actors) and container elements (e.g., packages and swimlanes). Modelling thinking on reuse of common assets helps to modularize the systems better and help to identify in the models what parts are related to mandatory or variable features.

```
1.  Variant { name : "A-W" for : "Automated Windows"
2.    + UseCase : "OpenAndCloseWinAuto" in Package : "WinMan"
3.    + Partition : "WinActuator" in ActDiagram :"CtrlTempAuto"
4.    ...
5.  }
6.  Variant { name : "R-H"
7.     for : And ("Remote Heating Ctrl","Automated Heating","Internet")
8.       + UseCase : "CtrlTempRemotely" in Package : "Heating"
9.     Includes from UseCase : "CtrlTempRemotely"   to UseCase(s) :
10.              "NotifyByInternet" and "OpenAndCloseWinAuto" and "AdjustHeaterValue"
11.       ...
12. }
```

Figure 3.5: Composition specification of variants A-W (associated to an atomic feature expression - AUTOMATED WINDOWS) and R-H (associated to a compound feature expression - AND ("REMOTE HEATING CTRL" , "AUTOMATED HEATING" , "INTERNET").

3.2.3 Create or Modify VML4RE Composition Specification

Figure 3.5 illustrates a composition specification that guides the specification of the transformation of requirements specifications of products in the Smart Home SPL. VML4RE is a textual language that allows associating *actions*[5], to combinations of features written as logic expressions that we call *feature expressions*. Feature expressions can be:

- *Atomic*, representing atomic features such as AUTOMATED WINDOWS in Figure 3.5, Line 1, and
- *Compound*, containing logic operators such as AND, NOT and OR such as AND ("REMOTE HEATING CTRL","AUTOMATED HEATING","INTERNET") in line 7 of Figure 3.5.

Feature expressions evaluation in a feature model configuration works as follows: if a feature is selected to be part of a product, that feature evaluates to TRUE; otherwise, if the feature is not selected it evaluates to FALSE. Thus, a feature expression can be evaluated to TRUE or FALSE based on the boolean value of each feature in the feature expression. If a feature expression evaluates to TRUE its corresponding actions will be processed and applied to the base model. Otherwise, if the feature expression evaluates to FALSE, the next feature expressions will be read and evaluated until the end of the composition specification.

In our example, if AUTOMATEDHEATING, REMOTE HEATING CTRL, AUTOMATED HEATING and INTERNET features are selected in a product configuration, the feature expression (i.e., the compound feature expression: And ("Remote Heating Ctrl","Automated Heating","Internet")) associated to the variant named “R-H” will be evaluated to TRUE. The consequence is that the actions in the “R-H” variant block (Figure 3.5, lines 6-12) will be processed and applied to a base model. For example, the CTRLTEMPREMOTELY use case will be inserted into the package HEATING and then it will be related to other

[5]Each *action* wraps a set of model transformations for specific requirements models such as use cases and activity diagrams.

use cases using Include and Extend relationships. If more than one feature expression is evaluated to TRUE, the default composition order follows a top-down sequence. Note that for simplicity of explanation we omitted, in Figure 3.5, some of the actions, such as the insertion of the actors WINSENSOR and WINACTUATOR and partitions such as HEATER.

Figure 3.3 (a) shows use case and activity diagrams fragments, such as actors and use cases, related with the variants shown in Figure 3.5. The base mechanism to relate parts of the model-based specifications to the features they design is to use a correspondence table (or mapping table), as presented by [48, 97, 30, 9, 45]. It is possible to parse the composition specification to generate the mapping between variants and parts of the model-based specifications (more details in Chapter 14). Therefore, for example, if the variant A-W inserts the OPENANDCLOSEWINAUTO use case, we relate the feature expression of A-W ("AUTOMATEDWINDOWS") to OPENANDCLOSEWINAUTO. To facilitate the visualization of such relationships with the models, we assign different gray tones to the model elements according to the feature expressions of variants that they are related to (see mapping in Figure 3.3 (b)). Also, note that specific model fragments could be related to more than one variant; this may be considered as a M-to-N (where M, N >= 1) mapping between feature expressions of variants and model fragments (not illustrated in Figure 3.3).

3.2.4 Check Consistency Using VCC

Consistency checking aims at ensuring that inconsistent models do not become part of the specifications of a given product. Our work aims at ensuring that all the models for products that can be derived from a feature model have consistent specifications, and that the feature model is consistent with its corresponding model-based specifications. This is achieved through the description and verification of consistency constraints of the relationships between a feature model and its corresponding model-based specifications.

At least two requirements, one inferred from the feature model and one inferred from the model-based specifications, that cannot be accomplished in at least one product, generates an inconsistency. From the models described in the previous section we can infer at least two requirements that cannot be achieved together in at least one product of the Smart Home. Requirement-1, which is inferred from the domain constraints expressed in the feature model and from their relationships with the feature expressions, and Requirement-2, inferred from the relationships between elements in the model-based specifications:

- *Requirement-1*: *Only one, none or both* R-H *and* A-W *variants can be included in a product*. This information is inferred from the feature model and the feature expressions because all the features in the feature expression of the R-H variant are optional and not exclusive between them (i.e., REMOTE HEATING CTRL, AUTOMATED HEATING and INTERNET are optional features), and the only feature in the feature expression of variant A-W is also optional (i.e., the AUTOMATED WINDOWS feature is optional).

- *Requirement-2*: *If the use case* CTRLTEMPREMOTELY *is provided in a product, then the use*

case OPENANDCLOSEWINAUTO *and related steps in the activity diagram must be provided too in order to support all the intended use scenarios*. This is implicit in the Include relationship from the use case CTRLTEMPREMOTELY to OPENANDCLOSEWINAUTO in the use case diagram in Figure 3.3 (a) (right-hand side) and also because of the control flows between the step: DETERMINE LOW ENERGY CONSUMPTION STRATEGY and OPEN WINDOWS / CLOSE WINDOWS in Figure 3.3 (a) (left-hand side).

In the particular case of the Smart Home use scenarios, a inconsistency can be detected: there is at least one product that cannot satisfy Requirement-1 and Requirement-2. Let's analyse how both requirements are satisfied or not in each product.

While Requirement-1 is satisfied by all the product configurations, therefore satisfied by Product-1 and Product-2, since they have the same feature model, Requirement-2 is satisfied by Product-1 only as its use cases and activities supported all the required use scenarios for this product. For example, given that the base use case CTRLTEMPREMOTELY was provided in Product-1, the use cases related to it through an Include relationship, for example OPENANDCLOSEWINAUTO, are also present in the model. The Include relationship supports the reuse of functionality in use case diagrams in which one use case (the *base* use case) requires the functionality of another use case (the *inclusion* use case). Therefore, all possible use scenarios related to CTRLTEMPREMOTELY are supported only when its *inclusion* use cases are included.

Requirement-2 is not satisfied by Product-2 because its feature configuration (shown in Figure 3.2 (c)) does not include the AUTOMATED WINDOWS feature. Therefore, the feature expression of variant A-W (i.e., AUTOMATED WINDOWS)(Figure 3.5, Line 1) evaluates to FALSE and the actions inside its variant block are not processed, for example, the inclusion of the use case OPENANDCLOSEWINAUTO. The result is that the functionality provided by OPENANDCLOSEWINAUTO will not be present in the requirements of Product-2 and therefore it will not be taken into account in later stages of its development process, thus given no support for the scenarios related to CTRLTEMPREMOTELY.

One solution to solve the inconsistency for our example would be to guarantee the presence of the feature AUTOMATED WINDOWS when AUTOMATIC HEATING or REMOTE HEATING CTRL are selected, in every possible feature model configuration. This can be guaranteed, for example, adding a domain constraint REQUIRES. Another solution is to establish that AUTOMATED WINDOWS will be a mandatory feature in the SPL. However, the number of possible feature combinations may grow exponentially with the number of features of the SPL. The result of this combinatory explosion makes it unfeasible to manually check the consistency of all products.

To guarantee that all the products derived from a feature model have consistent specifications, we take into account the relationships between constraints specified in feature models and its model-based specifications. This is an automatic process that we will describe in detail in Section 3.5. The result of this process is a report generated by our tool (see tool support in Section 3.6):

> "...Inconsistent use scenario(s) [CtrlTempRemotelly] and feature(s) in feature expression(s) of variant(s) [A-W], [R-H]. The Action: [Include from UseCase: CtrlTempRemotely to Use Case(s) OpenAndCloseWinAuto] implies a [Requires] relationship from variant [R-H] to required variant(s) [A-W] that is not enforced in the SPL feature model..."

Based on this information, developers may consider modifying the models to fix the inconsistency, for example:

- *Modify the feature model*: the set of SPL domain constraints that can be extracted from the feature model can be modified, for example by creating a REQUIRES relationship for AUTOMATEDHEATING feature to AUTOMATEDWINDOWS, or by changing the AUTOMATEDWINDOWS feature from optional to mandatory.
- *Modify use scenarios and the composition model:* for our particular rule, developers may want to check if indeed the Include association between use cases CTRLTEMPREMOTELY and OPENANDCLOSEWINAUTO is mandatory for every single product, or not.

3.2.5 Create Feature Model Configuration(s)

We presented two specific product configurations of the Smart Home SPL, one called Product-1 that includes all the SPL features (Figure 3.2 (b)) and another called Product-2 that has all features except AUTOMATED WINDOWS (Figure 3.2 (c)). It is possible to create more feature model configurations from the original feature model. For example, Figure 3.6 shows a feature model configuration for an economical smart home which does not contain any optional features except by AUTOMATED WINDOWS.

According to the SPLOT feature model analyzer it is possible to create 10 different valid configurations[6] for the feature model used in this example. However, for normal-sized examples the number of valid combinations of features in a feature model increases to millions (Chapter 4 - Validation shows examples). These numbers show that manual consistency checking is not a good option; an approach and tool support such as the one that we proposed in this work are very necessary.

3.2.6 Derive Model-Based Specifications for Product(s) Using VML4RE

In application engineering, the feature model configurations created in the previous subsection are used as a driver during the process to derive automatically product-specific model-based requirements specifications. The VML4RE interpreter first copies the initial model to a new model called the composed model or the *target model*. After that, it processes the actions whose features expressions evaluate to TRUE as explained in the Subsection 3.2.3.

[6] http://www.splot-research.org/

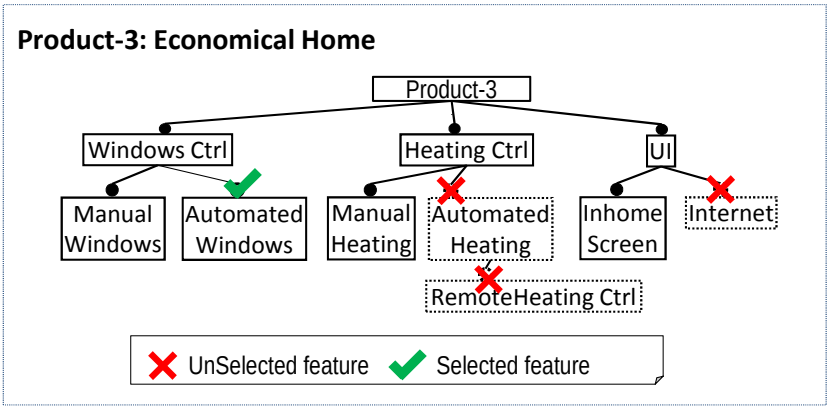


Figure 3.6: Sample configuration that excludes all the features except Automated Windows.

The first example of scenarios derivation was already presented in Figure 3.3 and corresponds to the feature model configuration shown in Figure 3.2 (b). Given the possibility of defining in a unique VML4RE specification the relationships between a feature model and several requirements models (e.g., use case and activity models), our interpreter produces different product-specific requirements models in the same process. In the exemplar feature model configuration shown in Figure 3.2 (b) all the features were selected, therefore, all the use cases, actors and activity diagrams, were added into the model.

Figure 3.6 shows the feature model configuration of an Economical Smart Home. The VML4RE interpreter processes the SPL requirements models and the feature model configuration, to derive a product-specific requirements model. In the economical Smart Home the actions in the R-H variant were not processed because its respective feature expression evaluated to FALSE. As explained before, if a feature is selected in the configuration, it evaluates to TRUE; otherwise, if it is not selected, it evaluates to FALSE. Therefore, the feature expression of the variant R-H: AND ("REMOTE HEATING CTRL", "AUTOMATED HEATING", "INTERNET") evaluates AND (FALSE, FALSE, FALSE) which evaluates to FALSE, while the variant A-W, evaluates to TRUE.

3.3 Main Elements

This section starts with definitions that complement some of the concepts presented informally in Chapter 2 - Background. Following to those definitions, this section describes the main elements of the abstract syntax and semantics of DCC4SPL. Sections 3.4 and 3.5 of this chapter build upon those main elements to describe the two main parts of DCC4SPL, VML4RE and VCC.

3.3.1 Preliminars

The abstract syntax of DCC4SPL can be specified by means of a metamodel. There are different concrete syntaxes to represent metamodels such as tree-based diagrams, class

diagrams and textual languages. In this dissertation we employ the class diagram notation as it is used frequently in MDD and most developers are familiar with it. A metamodel using a class diagram notation is represented by classes with attributes that are related using references. Classes in metamodels are usually referred in literature as *metaclasses*. References are further distinguished into containment references and non-containment references [85]:

- *Containment* is used to relate a parent model element and a child model element that is declared in the context of the parent element. An example can be found in the declaration of a method within the body of a class declaration in object-oriented programming (OOP) languages.

- *Non-containment* is used to relate a model element with an element that is declared elsewhere (not as one of its children). An example in OOP languages can be found in a method call (declared in a statement in the body of a method declaration) that relates to the method that it calls using a (non-containment) reference.

Model and metamodel, are defined based on the definition of Typed Graph as it is the most intuitive way to represent a graph with interrelated nodes. This subsection uses definitions from the literature whenever possible to guarantee readers not familiar with these concepts to understand them easily.

Typed Graph. A typed graph is a triple $\langle V, E, \tau \rangle$ where V is a finite set of vertices, $E \subseteq V \times V$ is a finite set of directed edges connecting the vertices and $\tau : \{V \cup E\} \rightarrow Type \cup \{containment, non-containtment\}$ is a typing function for the elements of V and E such that $\tau(v) \in Type$ if $v \in V$ and $\tau(e) \in \{containment, non-containtment\}$ if $e \in E$. Edges $(v, v') \in E$ are noted $v \rightarrow v'$. We furthermore impose that the graph $\langle V, \{v \rightarrow v' \in E | \tau(v \rightarrow v') = containment\} \rangle$ is acyclic[7]. The set of all typed graphs is called TG (adapted from [20]).

Typed Graph Instance. Let $\langle V, E, \tau \rangle = g$, $\langle V', E', \tau \rangle = g' \in TG$ be typed graphs. g' is a typed graph instance of g, written $g' \models g$, iff for all $v'_1 \rightarrow v'_2 \in E'$ there is a $v_1 \rightarrow v_2 \in E$ such that $\tau(v'_1) = \tau(v_1), \tau(v'_2) = \tau(v_2)$ and $\tau(v'_1 \rightarrow v'_2) = \tau(v_1 \rightarrow v_2)$. Notice that we only enforce that connections between vertices of g' must exist also in g and have the same type (taken from [20]).

Metamodel. A metamodel $\langle V, E, \tau \rangle \in TG$ is a typed graph where τ is a bijective typing function. The set of all metamodels is called $META$ (taken from [20]).

[7] According to Barroca et. al. [20] using containment and reference as types for edges allows to model the different types of associations between the elements of a metamodel or a model.

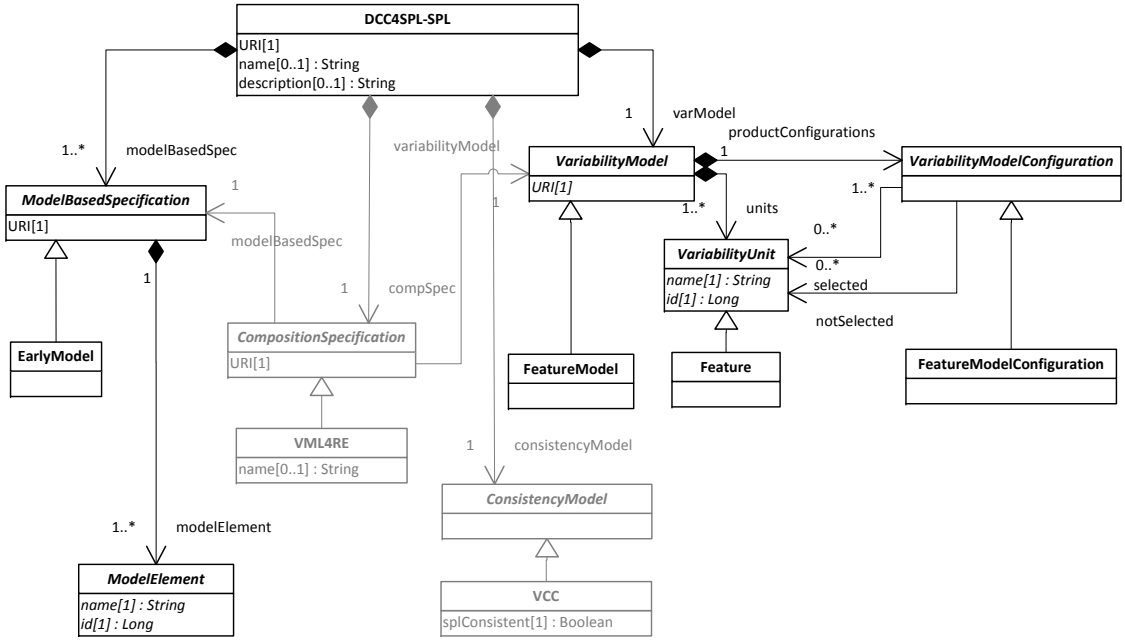


Figure 3.7: Main parts in the DCC4SPL metamodel.

Model. A model is a 4-tuple $\langle V, E, \tau, M \rangle$ where $\langle V, E, \tau \rangle$ is a typed graph. Moreover $M = \langle V', E', \tau' \rangle \in META$ is a Metamodel and the codomain of τ equals the codomain of τ'. Finally $\langle V, E, \tau \rangle \models M$, which means $\langle V, E, \tau \rangle$ is an instance of a metamodel M. The set of all models for a metamodel M is called $MODEL^M$ (taken from [20]).

Model Fragment. A model fragment mf_1 is a model fragment of another model fragment mf_2 if all model elements in mf_1 are also in mf_2 (taken from [74]).

3.3.2 Abstract Syntax

Figure 3.7 shows the main parts of the DCC4SPL metamodel. The abstract metaclasses Model-Based Specification, Composition Specification, Consistency Model and Variability Model, are specialized with Early Model, VML4RE, VCC and Feature Model, respectively. Figure 3.7 shows in grey the metaclasses that will be presented during the description of VML4RE and VCC in Sections 3.4 and 3.5, respectively.

The rest of this subsection presents the main metaclasses of the DCC4SPL metamodel, starting by the root metaclass DCC4SPL-SPL. This metamodel considers the concepts (metaclasses) it deals with, how they interrelate (using containment and non-containment references) and the properties (attributes) they have.

DCC4SPL-SPL. Represents an SPL created according to our approach for Derivation and Consistency Checking of models in early SPLE (DCC4SPL). DCC4SPL-SPL has a URI[8] and

[8]Uniform Resource Identifier (URI) is a compact string of characters for identifying an abstract or physical resource such as a model.

may have a name and a textual description. It also contains a Composition Specification, its corresponding Variability Model, one or more Model-Based Specifications[9] and a Consistency Model.

The rest of this section groups the metaclasses according to the metaclass where they are contained: Model-Based Specification (page 50) and Variability Model (page 55). Composition Specification (page 59) and Consistency Model (page 70) are described in Sections 3.4 and 3.5.

Model-Based Specification. Represents any of the models that specify behaviour, structure, or qualities of the SPL system. A Model-Based Specification has a URI, and contains one or more Model Elements.

Model Element. Represents an element contained in a Model-Based Specification. VML4RE employs this concept as an abstraction for model elements in model-based requirements specifications. For example, use cases and activity diagrams are UML-specific model-based specifications that contain Model Elements such as Package, Use Case, Activity Partition, Actor, Activity and Action.

Figure 3.8 shows the metaclass Element from UML that is referenced by the DCC4SPL metaclass Model Element. Element in UML serves as an adapter for all the model elements used in use cases and activity diagrams as well as almost all the UML metaclasses.

Early Model. Represents models written during requirements modelling and architecture design. VML4RE (Section 3.4) focuses on types of models that are most typically employed during requirements engineering, for example, use scenarios (or *scenarios*, for short).

Use Scenarios Model. Represents a set of scenarios which are essentially short histories composed of a list of steps. In UML, use cases and activity diagrams can be used to represent scenarios. Each use case describes how actors (i.e., persons, organizations or other (sub)systems) interact with the system to achieve a specific goal[10].

Figure 3.8 shows the relationships between the part of the DCC4SPL metamodel related to VML4RE and the part of the UML metamodel related to scenario modelling. The main metaclasses in UML related to scenarios are: Activity, Actor and Use Case. They, as well as other metaclasses that have concrete representations in use cases and activity diagrams (e.g., Activity Partition, Extend, Activity Node, Include, Package, Action), extend the metaclass Element.

Figure 3.8 shows in grey both the specialization hierarchy from Actor, Activity and Use Case to Model Element. That figure also shows that Behavior Classifiers, such as Use

[9] We usually refer to instances of a metaclass by quantifying its name. Hence, "DCC4SPL-SPL contains one or more Model-Based Specifications" means "An instance of the DCC4SPL-SPL metaclass contains one or more instances of the Model-Based Specification metaclass".

[10] A comprehensive study of the contemporary scenario-based work is presented in [1].

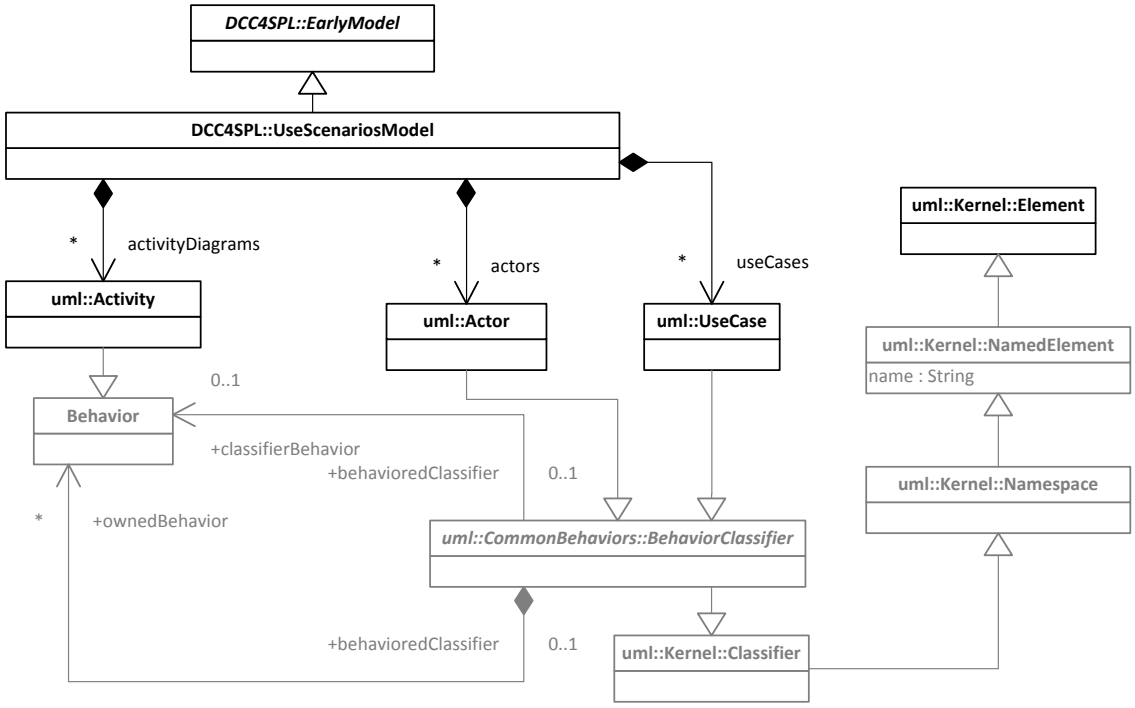


Figure 3.8: Some of the UML metaclasses related to scenario modelling and to the Use Scenarios Model and Model Element metaclasses of DCC4SPL.

Cases and Actors, relate to Activities through a containment relationship. This means that a Use Case can reference behaviour specifications modelled using Activities. Also, each Activity as a specialization of Behavior, can specify how its context classifier, i.e., a specific use case, changes state over time. Next we describe in more detail Activity and Use Case diagrams to help to understand their relationships.

Activity. Represents a major task that must take place in order to fulfil an operation contract. For example, the invocation of an operation, a step in a business process or an entire business process. An Activity groups all the model elements of an activity diagram such as ActivityNode, Action, ActivityEdge and ActivityPartition. Figure 3.9 presents part of the UML usually employed to model activity diagrams.

- *Activity Node* is an abstract metaclass that represents Control Nodes and Actions, which are connected by Activity Edges.
- *Control Node* is an abstract Activity Node that coordinates flows between nodes in an Activity. It covers Initial Node, Final Node, Fork Node, Join Node, Decision Node and Merge Node.
  - *Initial Node* is a control node where a flow starts when the activity is invoked.
  - *Final Node* is an abstract control node where a flow in an activity stops. *Activity Final Node* is a final node that stops all flows in an activity, while a *Flow Final*

Figure 3.9: Part of the UML metamodel focused on activity diagrams.

Node destroys all tokens that arrive at it and it has no effect on other flows in the activity.

  - *Fork Node* is a control node that splits a flow into multiple concurrent flows.
  - *Join Node* is a control node that synchronizes multiple flows.
  - *Decision Node* is a control node that chooses between outgoing flows. Which of the edges is actually traversed depends on the evaluation of the *guards* on the outgoing edges.
  - *Merge Node* is a control node that brings together multiple alternate flows. It is not used to synchronize concurrent flows but to accept one among several alternate flows.

- *uml::Action*[11] is an abstract metaclass that specializes Activity Node and that represents a single atomic step within an activity, i.e. which is not further decomposed within the activity. An Action is a named element that is the fundamental unit of executable functionality. Thus, the execution of an Action represents some transformation or processing in the modelled system.

- *Actvity Edge* is an abstract metaclass for directed connections between two activity nodes.

- *Activity Partitions* are groups of elements in an activity diagram.

Use Case. Represents and defines the interactions between actors and the system under consideration to accomplish a goal. Figure 3.10 presents a subset of the use cases diagrams metamodel that includes the elements that are related with VML4RE such as: Package, Use Case, Actor, Generalization, Include and Association.

- *Subject* metaclass represents the system under consideration to which the use cases apply. The required behaviour of the subject is specified by one or more Use Cases (metaclass), which are defined according to the needs of Actors.

- *Actor* metaclass are the users and any other systems that may interact with the subject. In other words, an Actor specifies a role played by a user or any other system that interacts with the Subject.

- *Extends* is a Directed Relationship from an extending use case to an extended use case that specifies how and when the behaviour defined in the extending use case can be inserted into the behaviour defined in the extended use case.

- *Include* is a Directed Relationship between two use cases, implying that the behaviour of the included use case is inserted into the behaviour of the including use case.

[11]The prefix *uml* means that we refer to the metaclass Action in the UML metamodel and not the metaclass Action in VML4RE.

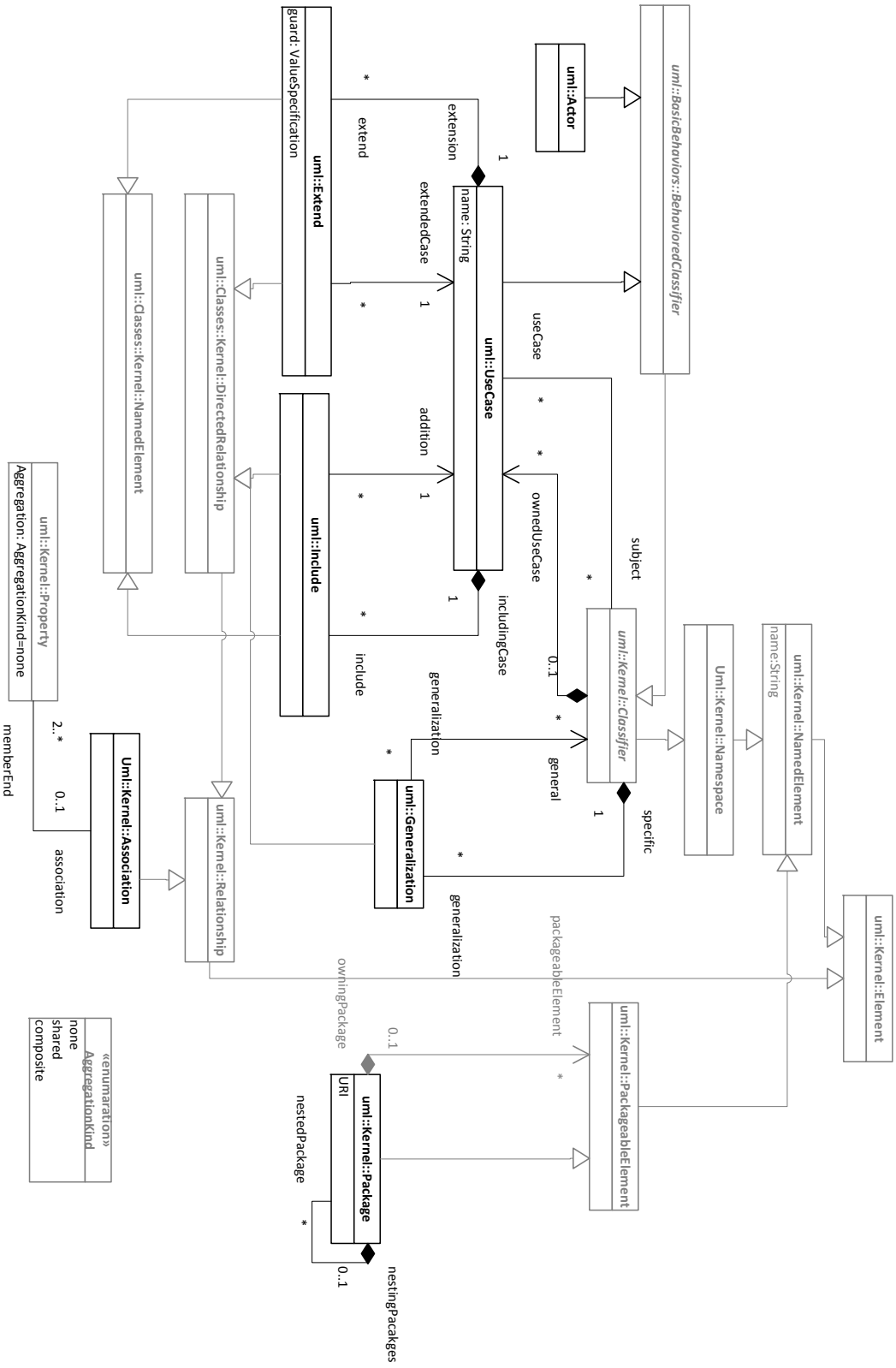


Figure 3.10: Part of UML focused on use case diagrams.

It is also a kind of Named Element so that it can have a name in the context of its owning use case. The including use case may only depend on the result (value) of the included use case. This value is obtained as a result of the execution of the included use case. Note that the included use case is not optional, and is always required for the including use case to execute correctly.

- *Namespace* is an element in a model that contains a set of named elements that can be identified by name.
- *Named Element* is an element in a model that has a name.
- *Classifier* is a Named Element that is a classification of instances; it means that it describes a set of instances that have features in common. It is possible to define Generalization relationships to other classifiers. A Classifier can specify a generalization hierarchy by referencing its general classifiers.
- *Package* is used to group elements and to provide a namespace for the grouped elements. Also, a package may contain other packages.
- *Use Case* is the specification of a set of actions performed by a system, which yields an observable result that is typically of value for one or more actors or other stakeholders of the system.

Variability Model. Represents the central artefact in variability modelling that describes what variability units a product may have and what constraints govern the selection of combinations of variability units for individual products [97]. Variability Model and Variability Unit metaclasses are used as adapters for specific techniques for variability modelling. A Variability Model has a URI and contains one or more Variability Units and Variability Model Configurations.

Variability Unit. Represents a property, a functionality or non-functional quality that can vary among the systems in an SPL. Variability Units may be represented explicitly as features in a feature model or more implicitly in a DSL [94], or in any other form that is convenient for modelling variability in a specific project [97].

Variability Model Configuration. Represents the information about which Variation Units were selected or not selected in a specific Variability Model during the application engineering activity. Therefore, a Variability Model Configuration references zero or more instances of selected Variability Units and zero or more instances of not selected Variability Units.

Feature Model Configuration. Represents a specific kind of Variability Model Configuration that keeps the information about the Features selected and not selected in a Feature Model.

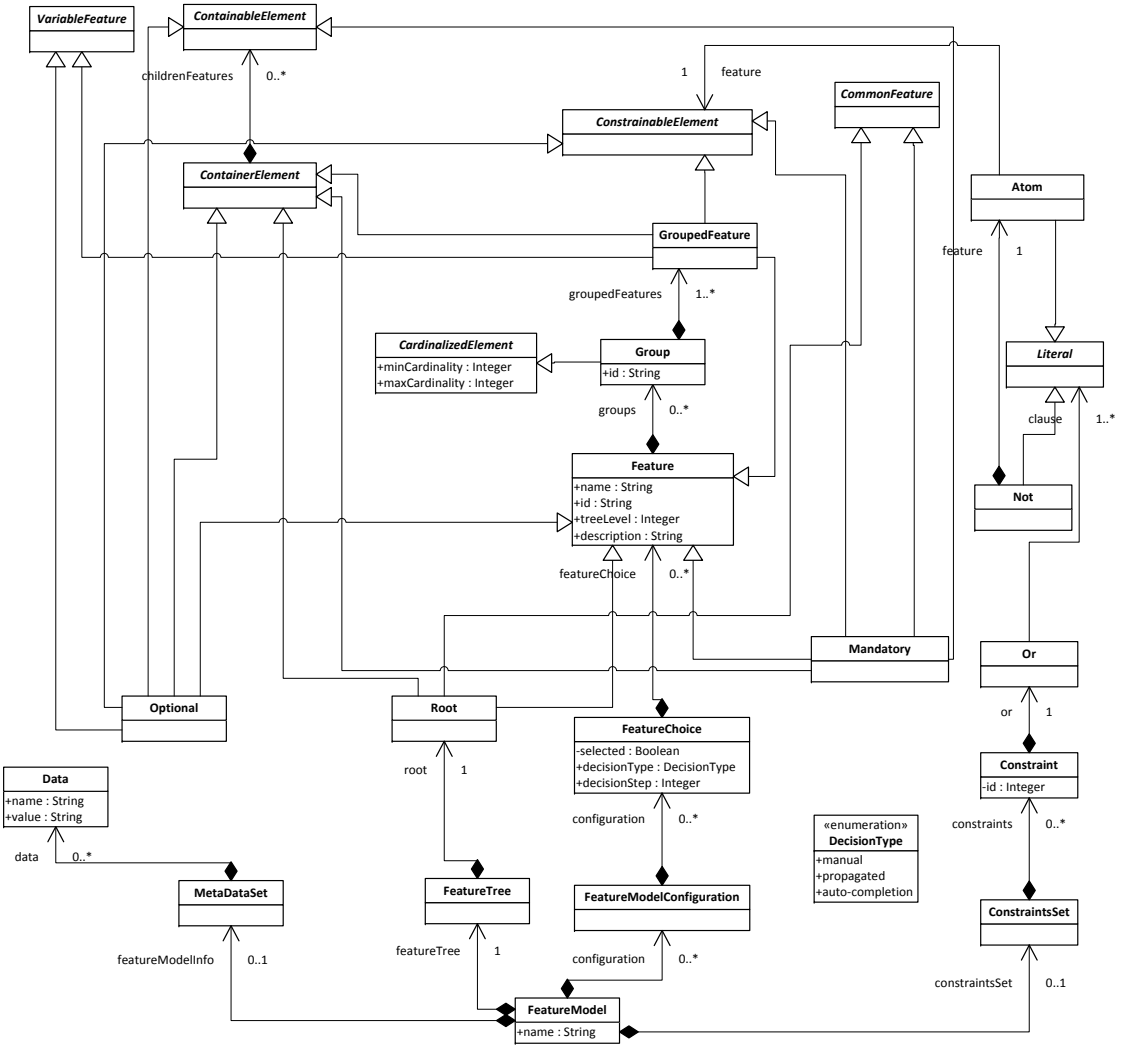


Figure 3.11: Metaclasses from SPLOT for feature models.

Feature Model. Represents a specific kind of Variability Model that characterizes all the products of an SPL in terms of Features. An instance of a Feature Model is visually represented by means of a feature diagram. Feature models are used during SPLE as input to produce other assets such as pieces of code. We contribute to the industry and research community with a translation to EMF Ecore of the SPLOT's SXFM [12] feature model specification (plugins and more documentation are in the SPLOT website[13]). Figure 3.11 presents the feature model metamodel whose main metaclass is Feature Model. It has a name (usually the same name of the SPL) and contains a Feature Tree, zero or more Feature Model Configurations and optionally, a Constraints Set and a Meta Data Set.

- *Meta Data Set* represents a set of Data fields that can be attached to a Feature Model to provide information of the types and values defined by Data instances.

[12]http://www.splot-research.org/: Software Product Line Online Tools.
[13]http://gsd.uwaterloo.ca:8088/SPLOT/sxfm.html : Mauricio Alférez SPLOT SXFM specification in EMF ECore.

- *Data* represents an attribute attached to a Feature Model to provide more information about its creation. Examples of useful data are: description, creator, address, mail, phone, website, organization, department, date, and the reference of any publication where the feature model was originally created.
- *Feature Tree* represents the main part of a feature diagram represented as a tree-like taxonomical structure where the leaves are features with no children.
- *Root* represents the unique feature that contains all the other Features. Root is common to all the products in the SPL.
- *Feature* represents product functionalities, properties or non-functional qualities [79]. Thus, specific products can be characterized in terms of features. Features may be common to all products or vary between products. Therefore, the terms *commonality* and *variability* are often used to denote the Common and Variable Features within an SPL, respectively. A Feature has an unique name and id, the number of its tree level, and an optional textual description.
- *Common Feature* represents a Feature that is used in several products of an SPL. A Common feature becomes Mandatory when its predecessors are selected in a feature model configuration.
- *Mandatory* represents a Feature that will always be included in a product variant if their parent feature is included in the product variant. Mandatory features are not part of variability models in the true sense, but serve to structure or document their parent feature.
- *Variable Feature* represents Optional and Grouped Features.
- *Optional* represents selectable features that are not directly part of a Group of Grouped Features.
- *Grouped Feature* represents selectable features in a Group of Grouped Features.
- *Cardinalized Element* represents the minimum and maximum number of Grouped Features that can be selected in a Group of Features.
- *Group* represents a set of Grouped Features and it is identified with an id.
- *Containable Element* represents a Feature that can be contained by other Feature as a children.
- *Container Element* represents a Feature that can contain children Features.
- *Constrainable Element* represents a Feature that can be referenced in cross-tree Constraints.
- *Constraints Set* represents a set of zero or more cross-tree Constraints.

- *Constraint* represents an arbitrary cross-tree Constraint that is established among features that are Constrainable Elements, i.e., Optional, Mandatory, and Grouped Features. To guarantee compatibility and simplicity of the implementation code when analysing the consistency of the feature models, the constraints follow a CNF format. A constraint formula in CNF form is a conjunction of clauses and each clause is a disjunction of literals. Also, any instance of *Or* must contain at least two clauses to make sense the creation of a disjunction. For example, Feature-A requires Feature-B can be represented with the CNF clause: ¬Feature-A ∨ Feature-B. Feature-A excludes Feature-B can be represented with the CNF clause: ¬Feature-A∨ ¬Feature-B.

3.3.3 Semantics

Much of the semantics of DCC4SPL was explained informally during the process overview (page 35), the example (page 39) and the abstract syntax (page 49) sections of this chapter. This section is dedicated to complement and harmonize those definitions with a more structured and rigorous operational semantics. The semantic specifications Consistency Checking Using VCC (page 73) and VML4RE Composition (page 66) complete the semantics of DCC4SPL.

Let $dSpl \models DCC4SPL\text{-}SPL$ (i.e., dSpl is an instance of DCC4SPL-SPL), $vml4re \models VML4RE$, $vml4re \in dSpl.compSpec$, $mbs \in dSpl.modelBasedSpec$, $fm \in dSpl.varModel$, $fmcSet = fm.productConfigurations$, $vcc \in dSpl.consistencyModel$ and $vcc \models VCC$. Moreover, let mbs, $vml4re$, $fm \neq \emptyset$. The DCC4SPL process is defined as follows:

$$\frac{mbs, fm, vml4re}{Consistency\,Checking\,Using\,VCC} \tag{3.1}$$

$$\frac{fmcSet, mbs, vml4re, [vcc.splConsistent] = TRUE}{\forall fmc_i |\, fmc_i \in fmcSet\ \ VML4RE\,Composition} \tag{3.2}$$

Equation 3.1 means that Consistency Checking Using VCC will be performed based on a model-based specification (*mbs*), a particular variability model such as a feature model (*fm*), and a composition specification (*vml4re*). Consistency Checking Using VCC is defined by Equations 3.5 and 3.6 in Section 3.5. Equation 3.2 means that Derive Model-Based Specifications for Products Using VML4RE (or VML4RE Composition, for short) will be performed if the SPL is consistent ($[vcc.splConsistent] = TRUE$). VML4RE Composition is defined by Equations 3.3 and 3.4 in Section 3.4 and it is performed for all the variability model configurations ($\forall fmc_i |\, fmc_i \in fmcSet$).

3.4 Inside VML4RE

VML4RE supports the DCC4SPL activities *Create or Modify VML4RE Composition Specification* and *Derive Model-Based Specifications for Products Using VML4RE* depicted in Figure 3.1. This section presents the abstract syntax, concrete syntax and semantics of VML4RE. We leave the description of the tool support to Section 3.6.

3.4.1 Abstract Syntax

Figure 3.12 shows the parts of the DCC4SPL metamodel that support VML4RE. To ease understanding of the metamodel, Figure 3.12 does not show all the specializations and details of VML4RE as they will be described during this section. Figure 3.12 presents in grey the parts already explained in Section 3.3 as well as the metaclasses related to the Consistency Model abstract metaclass. The specialization and details of the Consistency Model are related to the consistency checking capability of DCC4SPL that will be presented during the description of VCC in Section 3.5.

Composition Specification. Represents the description of how to derive the models for a specific product variant. A Composition Specification has a URI, and references a Model-Based Specification and a Variability Model.

VML4RE. Represents a composition specification that relates a Feature Model and an Early Model (a subtype of Model-Based Specification). VML4RE employs Variants that describe how the use scenarios are transformed into product-specific use scenarios, depending on a selection of features. VML4RE has an optional name and contains one or more Variants.

Variant. Represents the language construct that describes how model-based specifications are varied according to certain combination of features. A Variant has an id and optionally a name given by the developer. Also, a Variant contains one feature Expression and one or more Actions that will customize the SPL model-based specifications when the feature expression associated with the Variant evaluates to TRUE.

Expression. Represents the condition associated to each Variant that must be satisfied to trigger the execution of the Actions inside the Variant. An Expression can be either atomic or compound. Atomic, representing an expression composed of a single reference to a Variability Unit (i.e., a Variability Unit Ref). Compound, containing several Variability Unit Ref (each one referencing an existing Variability Unit, such as Feature) related by logic operators such as And, Not, Or and XOr. Instances of the Or, XOr and And metaclasses make sense when they contain at least two expressions; this is indicated by the cardinality of 2..* in the metamodel.

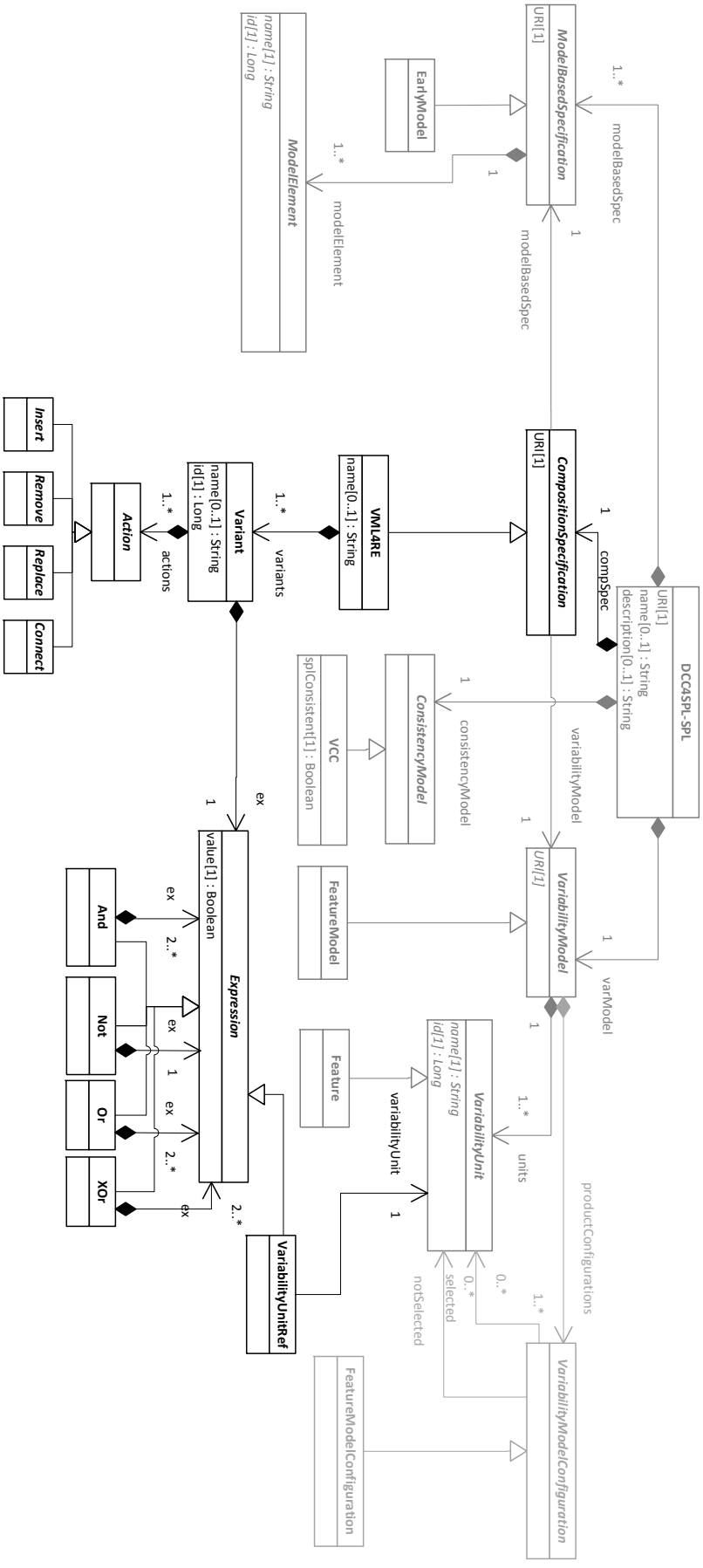


Figure 3.12: Parts of the DCC4SPL metamodel related to VML4RE.

The composite pattern was used here to model the metaclasses Expression, Variability Unit Ref, Not, And, Or and XOr. The intent of this pattern is to compose objects into tree-like structures to represent part-whole hierarchies. The composite pattern allows the uniform treatment of atomic expressions and compound expressions.

To understand the concept of an Expression, we provide a glimpse of the concrete syntax for And, Not, Or and XOr (this will be further described in Section 3.4.3). A textual representation of an instance of these expressions written in prefix notation may have the form:

AND (EXPRESSION$_i$, EXPRESSION$_{i+1}$,..., EXPRESSION$_n$)

NOT (EXPRESSION)

OR (EXPRESSION$_i$, EXPRESSION$_{i+1}$,..., EXPRESSION$_n$)

XOr (EXPRESSION$_i$, EXPRESSION$_{i+1}$,..., EXPRESSION$_n$)

Where i =1; i<n; i and n are natural numbers, and n is the number of subexpressions.

Variability Unit Ref. Represents an atomic Expression that references an existing Variability Unit, for example, a Feature.

And. Represents a compound Expression that evaluates to TRUE when all its contained expressions evaluate to TRUE.

Not. Represents a compound Expression that evaluates to TRUE when its contained expression evaluates to FALSE.

Or. Represents a compound Expression that evaluates to TRUE when at least one of its contained expressions evaluates to TRUE.

XOr. Represents a compound Expression that evaluates to TRUE when only one of its contained expressions evaluates to TRUE.

Action. Represents the description of transformations of SPL Model-Based Specifications. Actions can add, update or remove model elements as well as add, update or remove links between existing or newly added model elements. VML4RE offers implementations of actions that apply to use case and activity diagrams. These actions are depicted in Figure 3.13. For example, ConnectActorUseCase connects an actor to an use case using an association; RemovePackage removes a package and its contained model elements; and InsertUseCase adds an use case in a use case diagram. For activity diagrams, actions such as ReplaceActivity replaces a generic activity by the activities from other activity diagram without including its initial and end nodes, and InsertActivityDiagram adds a complete activity diagram into the model-based specifications of a particular product.

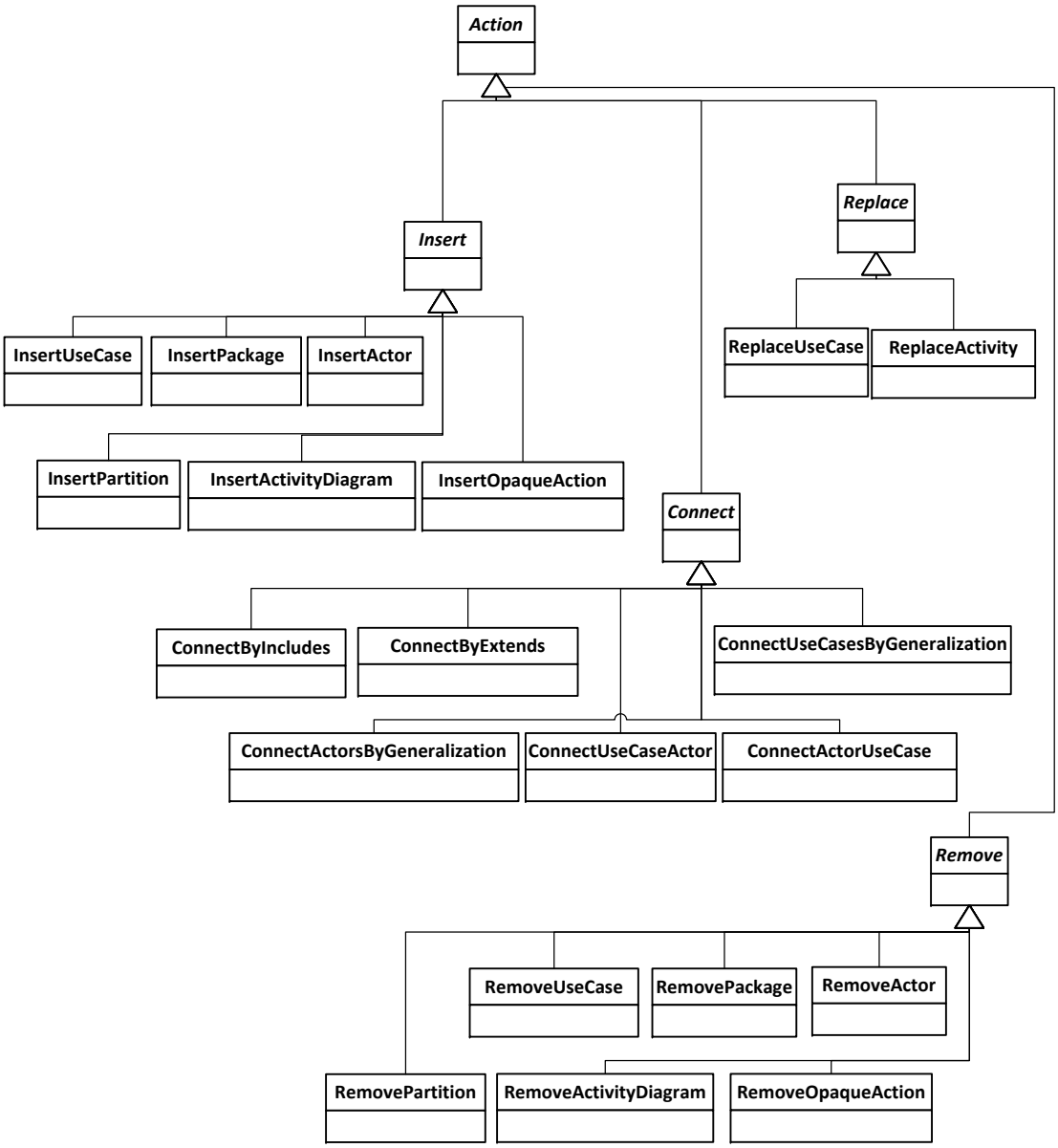


Figure 3.13: Actions in VML4RE.

```
1  VML4RE ::= "VML4RE" "{"
2    ("name" ":" name ['"','"'])?
3      "featuresPath" ":" varModel['"','"']
4      "useScenariosPath" ":" modelBasedSpec['"','"']
5      variants+
6  "}"
7  ;
8  Variant ::= "Variant" "{"
9    ("name" ":" name ['"','"'])?
10     "for" ":" ex
11     actions+
12 "}";
```

Figure 3.14: First part of the concrete syntax specification of VML4RE related to VML4RE and Variant.

3.4.2 Syntactic Mapping

Syntactic mapping relates the concepts modelled as metaclasses in a metamodel to a concrete syntax. The syntactic mapping for the VML4RE metaclasses that we want to represent consists on syntax rules. Each rule specifies the presentation of the text that represents instances of its related metaclass. Rules have two sides, left-hand side and right-hand side separated by the symbol "::=". The left-hand side denotes the name of the metaclass, while the right-hand side defines its syntax.

If a metaclass has attributes, we can specify a syntax for their values. To do so, we add brackets after the name of the attribute: OURMETACLASSWITHATTRIBUTE ::= OURATTRIBUTE[]; . Optionally, one can specify the name of a token inside the brackets: OURMETACLASSWITHATTRIBUTE ::= OURATTRIBUTE [MY_TOKEN]; . If the token name is omitted (i.e., OURATTRIBUTE[]), we mean the token TEXT, which is a predefined token that includes alphanumeric characters. Another possibility to specify the token definition that can be used to match the text for the attribute value is to do it in-line, for example: ['"','"'] allows to write arbitrary characters between the initial and final quote character for the value of attributes.

If a metaclass has a containment reference, we can do it like this: OURCONTAINER-METACLASS ::= "CONTAINER" OURCONTAINMENTREFERENCE;. It allows to represent instances of OURCONTAINERMETACLASS using the keyword "CONTAINER" followed by one instance of the type that OURCONTAINMENTREFERENCE points to.

If zero or more children need to be contained, the following rule is used: OURCONTAINERMETACLASS ::= "CONTAINER" OURCONTAINMENTREFERENCE* ;. Similarly, if one or more children need to be contained, the "*" sign can be replaced by "+". Next we will describe the concrete syntax of the main metaclasses in VML4RE that apply the constructs explained previously.

VML4RE and Variant. Figure 3.14 shows the first part of the syntax rules for VML4RE related to the metaclasses VML4RE and Variant. Figure 3.14, lines 2-4 and 9 shows that it is allowed to write arbitrary characters between the initial and final quote character for the value of attributes name, varModel, and modelBasedSpec.
Almost all the references in VML4RE are containment references. Figure 3.14, line 5 shows that a VML4RE model can have one or more variants. Also, Figure 3.14, line 10 shows that each variant can have only one feature expression and line 11 shows that each variant can have one or more actions.

Action. Figure 3.15 shows the second part of the concrete syntax that is related to the specializations of the metaclass Action. Lines 13 to 15 show the "?" sign which can be used to express that zero or one children need to be contained. This means that the last part of the rules, i.e., "("in Package :" inPkg['"','"'])?", is optional and therefore to specify an insertion of UseCase, Package or Actor it is not necessary to write the package name if we want to insert them in the main use case diagram.

Model Elements and Expression. Figure 3.16 shows the third part of the concrete syntax that is related to the specializations of some model elements and expresssions. Lines 42 to 45 show the rules for the metaclasses And, Or, XOr and Not. In these rules we define a prefix notation. We employ prefix notation for compatibility with previous versions of VML4RE and to abbreviate long feature expressions. Using " "(" ex "," ex ")" " at the begining of the And, Or and XOr rules forces at least two appearances of children expressions (Figure 3.16, lines 42 to 44).

Relationship between Concrete and Abstract Syntaxes. We include an example to show the dependencies between abstract and concrete syntaxes, and models that can conform to them. Figure 3.17 shows part of a composition specification written using VML4RE. The dashed arrows represent the dependency relationships between metaclasses in the abstract syntax (the metamodel) (Figure 3.17 (a)), rules in the concrete syntax (Figure 3.17 (b)), the textual representation that conforms to the grammar specified by the concrete syntax (Figure 3.17 (c)), and an object diagram that represents one possible representation of the model elements and their relationships related to the composition specification (Figure 3.17 (d)).

Figure 3.17 focuses on the specification of a Variant (metaclass) with name (attribute) "R-H". Variant R-H has a feature Expression (abstract metaclass) of type And (metaclass) composed of three atomic features expressions: VariabilityUnitRef (metaclass) "Remote Heating Control", VariabilityUnitRef (metaclass) "Automated Heating" and VariabilityUnitRef (metaclass) "Internet". Also, Figure 3.17 shows one Action (abstract metaclass) of type InsertActivityDiagram (metaclass) that is represented by the token "+ ActDiagram :" in the concrete syntax.

```
13 InsertUseCase ::= "+ UseCase :" newUc['"','"'] ("in Package :" inPkg['"','"'])? ;
14 InsertPackage ::= "+ Package :" newPkg['"','"'] ("in Package :"  inPkg['"','"'])? ;
15 InsertActor ::= "+ Actor :" newActor['"','"'] ("in Package :"  inPkg['"','"'])? ;
16 InsertPartition ::= "+ Partition :" newPartition['"','"']
   "in ActDiagram :"  inActivityDiagram['"','"'] ;
17 InsertActivityDiagram ::= "+ ActDiagram :" newActivityDiagram['"','"'] ;
18 InsertOpaqueAction ::= "+ OpaqueAction :" newOpaqueAction['"','"']
   "in ActDiagram :" inActDiagram['"','"'] ;
19
20 ConnectByIncludes ::= "Includes from UseCase :" fromUseCase['"','"']
   "to UseCase(s) :"  toUseCase['"','"'] ("and"  toUseCase['"','"'])*  ;
21 ConnectByExtends ::= "Extends from UseCase :" fromUseCase['"','"']
   "to UseCase(s) :"  toUseCase['"','"'] ("and"  toUseCase['"','"'])* ;
22 ConnectUseCasesByGeneralization ::= "Generalize from UseCase :" fromUseCase['"','"']
    "to UseCase :" toUseCase['"','"']  ;
23 ConnectActorsByGeneralization ::= "Generalize from Actor :" fromActor['"','"']
   "to Actor :" toActor['"','"']  ;
24 ConnectActorUseCase ::= "Associate from Actor :" fromActor['"','"']
   "to UseCase(s) :" toUseCase['"','"'] ("and"  toUseCase['"','"'])*  ;
25 ConnectUseCaseActor ::= "Associate from UseCase :"  fromUseCase['"','"']
   "to Actor(s) :" toActor['"','"']  ("and"  toActor['"','"'])*  ;
26
27 ReplaceActivity ::= "-+ Activity :"  genericActivity['"','"']
   "by ActDiagram :" byActivityDiagram['"','"'] ;
28 ReplaceUseCase ::= "-+ UseCase :" genericUseCase['"','"']
   "by UseCase :" bySpecificUseCase['"','"'] ;
29
30 RemoveUseCase ::= "- UseCase :" oldUc['"','"'] ("in Package :" inPkg['"','"'] )?  ;
31 RemovePackage ::= "- Package :" oldPkg['"','"'] ("in Package :" inPkg['"','"']  )? ;
32 RemoveActor ::= "- Actor :" oldActor['"','"'] ("in Package :" inPkg['"','"']  )? ;
33 RemovePartition ::= "- Partition :"  oldPartition['"','"']
   ("in ActDiagram :" inActivityDiagram['"','"']  )? ;
34 RemoveActivityDiagram ::= "- ActivityDiagram :"  oldActivityDiagram['"','"']  ;
35 RemoveOpaqueAction ::= "- OpaqueAction :" oldOpaqueAction['"','"']
   ("in ActDiagram :" inActDiagram['"','"']  ("and"  inActDiagram['"','"'])*)?;
```

Figure 3.15: Second part of the concrete syntax specification of VML4RE related to actions.

```
36  Pkg ::= name['"','"'] ;
37  UseCase ::= name['"','"'] ;
38  Partition ::= name['"','"']  ;
39  ActivityDiagram ::= name['"','"'] ;
40  Actor ::= name['"','"'] ;
41  OpaqueAction ::= name['"','"'] ;
42  And ::= "And"  "("( ex "," ex )("," ex )* ")"  ;
43  Or ::= "Or"  "(" ( ex "," ex )("," ex )* ")"  ;
44  XOr ::= "XOr"  "(" ( ex "," ex )("," ex )* ")"  ;
45  Not ::= "Not"  "("  ex  ")"  ;
46  VariabilityUnitRef ::= variabilityUnit['"','"'];
```

Figure 3.16: Third part of the concrete syntax specification of VML4RE related to ModelElement and Expressions.

3.4.3 Semantics

This section describes the operational semantics of VML4RE. It is organized by definitions ordered according to the precedence of the concepts.

Feature Model Configuration. A feature model configuration fmc is a 2-tuple $\langle selected, \overline{selected} \rangle$ where $selected$ and $\overline{selected}$ are respectively the set of selected and not-selected features of a system. Let $FSet$ be the set of features of a feature model, such that $selected, \overline{selected} \in FSet$, $selected \cap \overline{selected} = \emptyset$ and $selected \cup \overline{selected} = FSet$. Let $conf \in featureModel.productConfigurations$, we use the terms $conf.selected$ and $conf.\overline{selected}$ to respectively refer to the set of selected and not-selected features of $conf$, and $\emptyset_{conf}$ to denote the empty configuration $\langle \emptyset, \emptyset \rangle$ (adapted from [22]).

Derive Model-Based Specifications for Products Using VML4RE. Let $vml4re \models VML4RE$, $\{input, output\} \models UseScenariosModel$, $\{input, output\} \in vml4re.modelBasedSpec$, $fmc_1 \models featureModelConfiguration$, $fmc_1 \in vml4re.variabilityModel.productConfigurations$, $var \models Variant$ and $var \in vml4re.variants$. Moreover, let $ASet$ be the set of actions to be applied to $input$ to obtain $output$, $vml4re.variants \neq \emptyset$, $fmc_1 \neq \langle \emptyset, \emptyset \rangle$, $ASet = \emptyset$, $input \neq \emptyset$ and $output = \emptyset$. Derive Model-Based Specifications for Products Using VML4RE (or VML4RE Composition, for short) is performed product by product and for all the products of the SPL (i.e., $\forall fmc_i | \, fmc_i \in vml4re.variabilityModel.productConfigurations$). VML4RE Composition is defined as follows:

$$\frac{fmc_1, vml4re, [var.expression] = TRUE}{ASet = ASet \amalg var.actions} \tag{3.3}$$

$$\frac{input, ASet \neq \emptyset}{[ASet] = output} \tag{3.4}$$

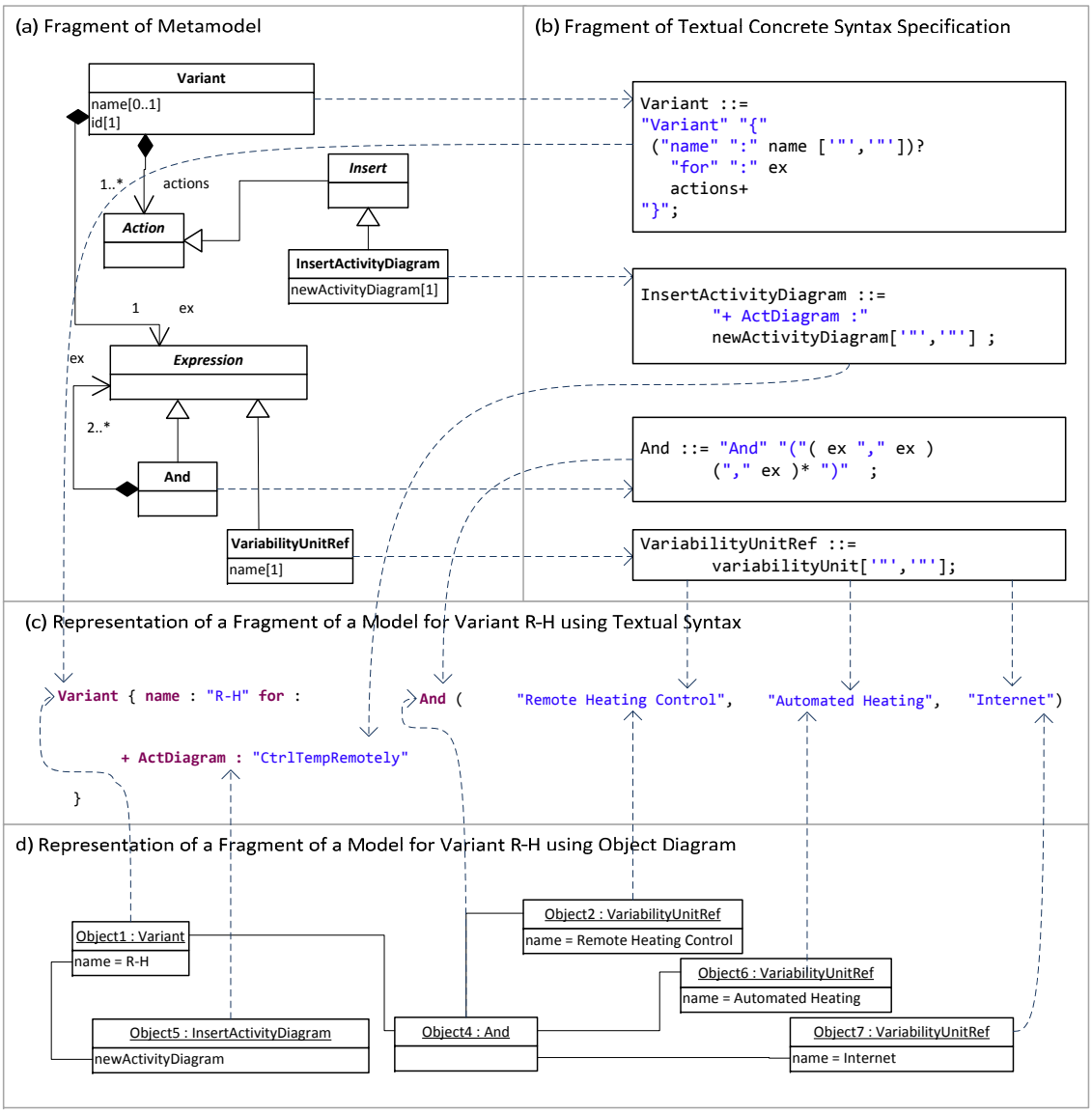


Figure 3.17: (c) Sample VML4RE model fragment related to the variant remote heating control in the Smart Home and its corresponding: (a) metamodel, (b) concrete syntax specification, and (d) relationships between model elements related to the sample model fragment.

Expression Evaluation. Equation 3.3 shows that the *expression* of each *variant* contained in the *vml4re* model will be evaluated. In case that the expression evaluates to TRUE, the actions of the variant will be added to $ASet$.

The value of an expression depends on the values associated with its identifiers. In VML4RE the identifiers are the names of features in the feature model configuration. If a variability unit name (e.g., a feature name) is selected in the configuration, we assign TRUE, otherwise we assign FALSE. Let $ex \models Expression$, which can be of different types VariabilityUnitRef, Not, And, Or, XOr. The evaluation of ex depends on its type as it is shown next:

$$[ex] = \begin{cases} [VariabilityUnitRef] & if\ ex \models VariabilityUnitRef \\ [Not] & if\ ex \models Not \\ [And] & if\ ex \models And \\ [Or] & if\ ex \models Or \\ [XOr] & if\ ex \models XOr \end{cases}$$

$$[VariabilityUnitRef] = \begin{cases} TRUE & if\ (VariabilityUnitRef.variabilityUnit \subseteq selected) \\ FALSE & if\ (VariabilityUnitRef.variabilityUnit \subseteq \overline{selected}) \end{cases}$$

$$[Not] = \begin{cases} TRUE & if\ [ex] = FALSE \\ FALSE & if\ [ex] = TRUE \end{cases}$$

$$[And] = \begin{cases} TRUE & if\ \forall ex_i \in And.ex \,|\, [ex_i] = TRUE \\ FALSE & if\ \exists ex_i \in And.ex \,|\, [ex_i] = FALSE \end{cases}$$

$$[Or] = \begin{cases} TRUE & if\ \exists ex_i \in Or.ex \,|\, [ex_i] = TRUE \\ FALSE & if\ \forall ex_i \in Or.ex \,|\, [ex_i] = FALSE \end{cases}$$

$$[XOr] = \begin{cases} TRUE & if\ \exists ex_i \in XOr.ex \,|\, [ex_i] = TRUE, \forall ex_j \in XOr.ex \,|\, [ex_j] = FALSE, i \neq j. \\ FALSE & Otherwise \end{cases}$$

Action Evaluation. Equation 3.4 shows that given a non-empty set of *input* and an *ASet*, the actions in *ASet* will be evaluated to produce the *output*. The *output* is a customized set of use scenarios for a specific product. We use graph transformations to express how each particular *Action* can transform the input models. Next we define graph transformation and after that we continue describing Action Evaluation.

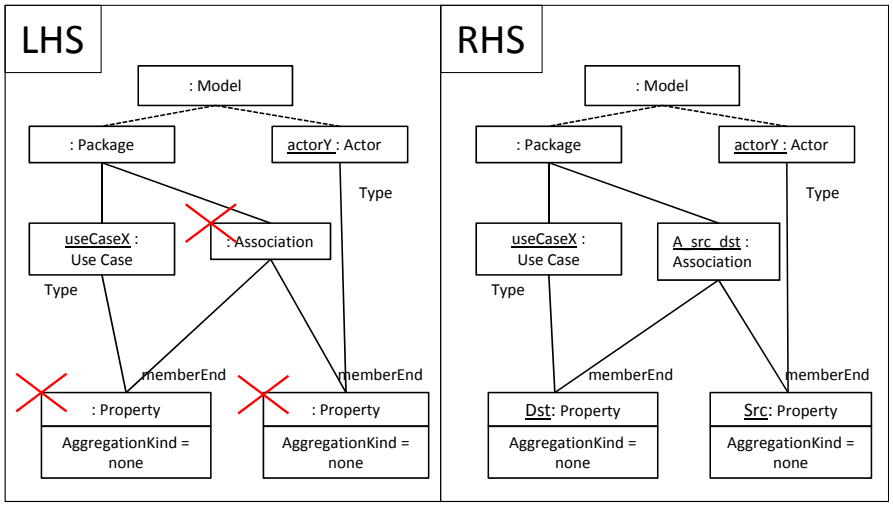


Figure 3.18: Graph rule to insert an Association between actorY and useCaseX

Graph Transformation. In general, a graph transformation is a graph rule r: L —› R for LHS graph L to a RHS graph R. The process of applying r to a graph G involves finding a graph monomorphism, h, from L to G and replacing h(L) in G with h(R) [81]. The notation used to express our graph transformations in the next definitions in this subsection is similar to the one used by [69] where the LHS and RHS patterns are denoted by a generalized form of object diagrams. However, for visual simplicity we added dashed lines between elements to represent any number of containments (in this case, package's containments). To the readers interested in details of this notation, please consult [69].

Action Evaluation (continuation). The semantics of each VML4RE action can be defined in terms of a model-to-model transformation. For instance, the action "ASSOCIATE FROM USECASE : *"useCaseX"* TO ACTOR : *"actorY"* " , connects the *useCaseX* using an association link to the *actorY*. The intended transformation of the use case diagram can be presented by the left hand side (LHS) and right hand side (RHS) graphs shown Figure 3.18, where the inputs are a use case model, a use case, and an actor. If there is already an association between the actor and the use case in the same package, the transformation is not applied to avoid duplicates. This is expressed with the cross in some elements in the LHS graph that act as negative application conditions (NAC). It means that any match against the LHS graph cannot have a package with any existing association between *useCaseX* and *actorY*.

Figure 3.19 illustrates the operator *replace* using the example "Replace Use Case". A replace in this context includes to remove a use case (indicated by the "-") and then insert a new use case (indicated by the "+") linked in the place of the old use case. For example, "-+ UseCase : useCaseB by UseCase : useCaseC".

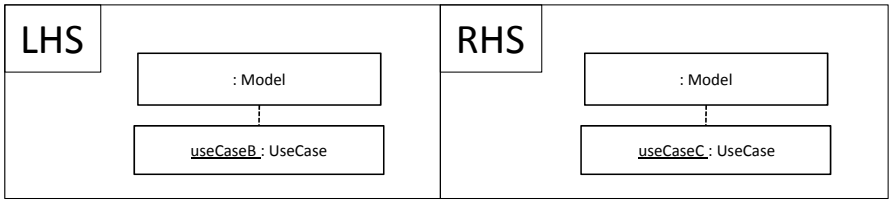


Figure 3.19: Graph rule to replace useCaseB by useCaseC.

3.5 Inside VCC

Variability Consistency Checking (VCC) supports the activity Consistency Checking Using VCC of the DCC4SPL process depicted in Figure 3.1. This section defines the metamodel and semantics of VCC while Section 3.6 describes tool support.

3.5.1 Abstract Syntax

Figure 3.20 shows the parts of the DCC4SPL metamodel that support VCC. This metamodel has a main abstract metaclass called Consistency Model that DCC4SPL specializes with VCC. This metamodel shows in grey some of the metaclasses related to the other parts of DCC4SPL to help readers to understand the relationships between the different parts.

Consistency Model. Represents a model used for consistency checking of the Variability Model (e.g., a Feature Model) against Model-Based Specifications. A Consistency Model contains an instance of Domain Constraints and Mapping Model, zero or more Consistency Rule Instances and optionally, a Model Elements Relationships Model.

VCC. Represents a Variability Consistency Checking model.

Domain Constraints. Represents a model containing the constraints between Variability Units which are obtained from Variability Model. Therefore, there is only one instance of Domain Constraints in all the Consistency Model.

Model Elements Relationships Model. Represents a model that records the information about dependencies and incompatibilities between model elements in Model-Based Specifications. This is an optional component in a Consistency Model that is useful for approaches that use that information to create Model-Based Specification Constraints based on the relationships of those model elements with Variants and Variability Units recorded in the Mapping Model.

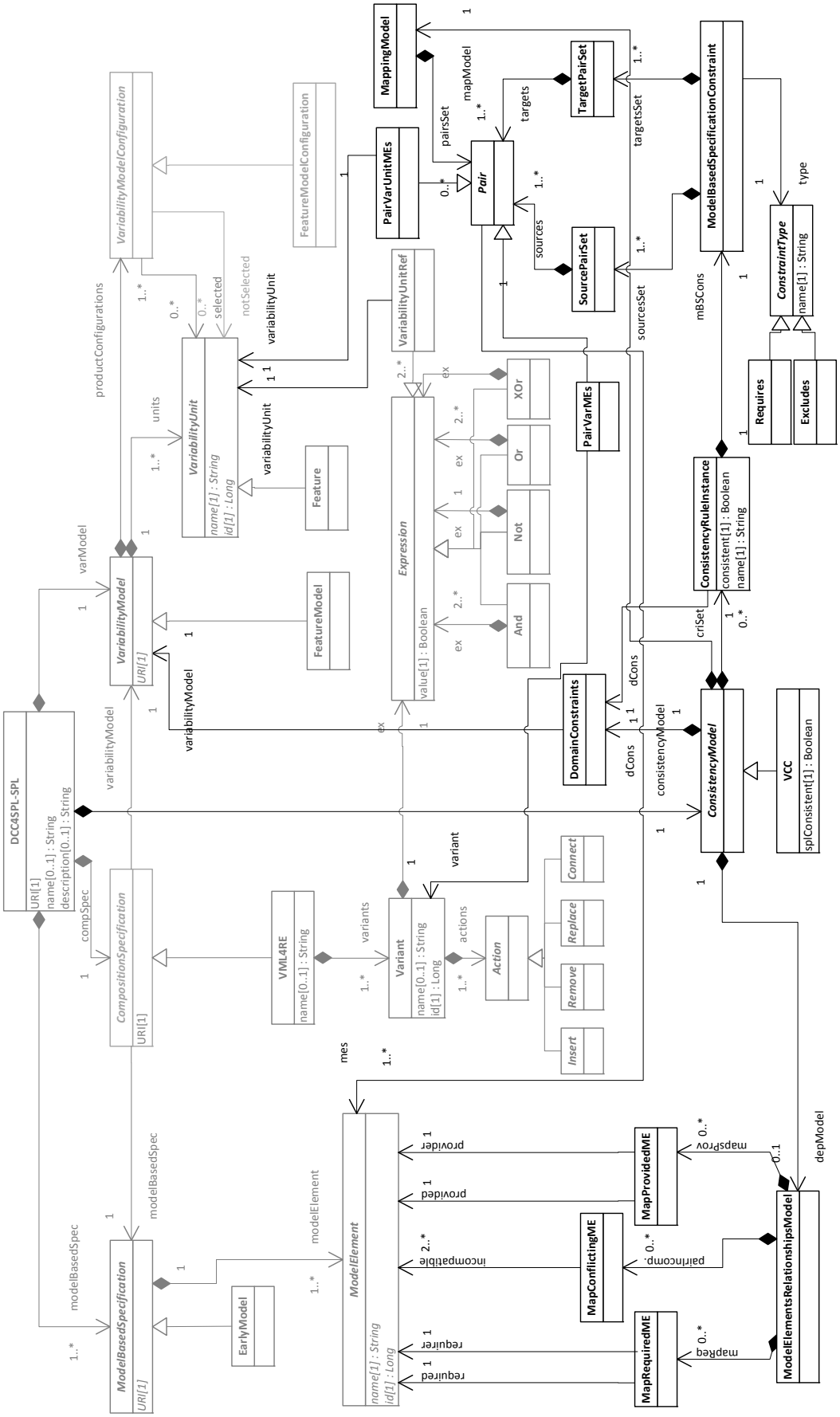


Figure 3.20: Parts of DCC4SPL metamodel related to VCC.

Map Required ME. Represents a 2-tuple $\langle required, requirer \rangle$ where the requirer Model Element depends on the required Model Element to be in the Model-Based Specification.

Map Provided ME. Represents a 2-tuple $\langle provided, provider \rangle$where the provider Model Element supplies the provided Model Element to the Model-Based Specification.

Map Conflicting ME. Represents a set of instances of Model Element that are incompatible between them when they are in the same Model-Based Specification.

Pair. Represents an abstract 2-tuple that references a set of Model Elements.

Mapping Model. Represents the set of Pairs.

Pair Var MEs. Represents a 2-tuple $\langle variant, mes \rangle$ that relates a Variant with a set of one or more Model Elements that depends on that Variant.

Pair Var Unit MEs. Represents a 2-tuple $\langle variabilityUnit, mes \rangle$ that relates a Variability Unit with a set of one or more Model Elements that depends on that Variability Unit.

Consistency Rule Instance. Represents one particular occurrence in a Model-Based Specification of a pattern described by a Model-Based Specification Constraint which needs to be evaluated against the Domain Constraints. A Consistency Rule Instance has a name (it corresponds to the name of the related Consistency Type plus an identifier), references the Domain Constraints instance and contains one Model-Based Specification Constraint.

Model-Based Specification Constraint. Represents a 2-tuple $\langle sourcesSet, targetsSet \rangle$ where sourcesSet contains instances of Source Pair Set and targetsSet contains instances of Target Pair Set. A Model-Based Specification Constraint defines well-formedness and design patters that need to be enforced in the Model-Based Specification.

Source Pair Set. Represents a set of instances of Pairs that contains a list of references to model elements that cause the creation of the Consistency Rule Instance.

Target Pair Set. Represents a set of instances of Pairs that contains a list of references to model elements that conflict or require model elements in the Source Pair Set.

Constraint Type. Represents the type of the Model-Based Specification Constraint. The concrete types considered in VCC are Requires and Excludes, they are described next[14].

[14]This metaclass could be optionally designed as an enumeration type whose fields consist of a fixed set of constants (REQUIRES and EXCLUDES). We choose this design to better meet our definition of metamodel which does not explicitly mention enumeration types in metaclasses.

Requires. Represents a type of constraint that implies that a Pair requires of other Pair.

Excludes. Represents a type of constraint that implies that a Pair is incompatible with other Pair.

3.5.2 Semantics

This section describes the operational semantics of VCC. It is organized by definitions ordered according to the precedence of the concepts.

Consistency Checking Using VCC. Let $vcc \models VCC, cris = vcc.criSet$ and $d = vcc.dCons$. Consistency checking using VCC is defined as follows:

$$\frac{vcc, cris, d, (\forall c | c \in cris, [c.consistent] = TRUE)}{vcc.splConsistent = TRUE} \tag{3.5}$$

$$\frac{vcc, cris, d, (\exists c | c \in cris, [c.consistent] = FALSE)}{vcc.splConsistent = FALSE} \tag{3.6}$$

Equations 3.5 and 3.6 show that all the instances of Consistency Rule Instance ($\forall c | c \in cris$), need to be evaluated for consistency. In case that all the instances are consistent ($\forall c | c \in cris, [c.consistent] = TRUE$), the SPL is consistent ($vcc.splConsistent = TRUE$). If any Consistency Rule Instance is not consistent ($\exists c | c \in cris, [c.consistent] = FALSE$), the SPL is inconsistent ($vcc.splConsistent = FALSE$) .

Printing Consistency Checking Results. The results that VCC prints are the information about the violated consistency rules found in an SPL. This information includes: (1) the name of the violated consistency rule, (2) the name of the variant (or variability unit), and (3) the name and type of the model elements involved. Printing Consistency Checking Results is defined as follows:

$$\frac{c \in cris, [c.consistent] = FALSE}{print(c)} \tag{3.7}$$

Equation 3.7 shows that only instances of Consistency Rule Instance evaluated as inconsistent ($[c.consistent] = FALSE$) will be printed. $Print(c)$ is defined as follows:

$$\frac{ss \in c.mBSCons.sourcesSet,\ ts \in c.mBSCons.targetsSet}{print(c.name), print(ss),\ print(ts)} \tag{3.8}$$

Equation 3.8 shows that printing a particular instance of Consistency Rule Instance will print its name ($print(c.name)$) as well as its source set ss and target set ts

($print(ss)$, $print(ts)$). $Print(ss)$ is defined as follows:

$$\frac{sp \in ss.sources,\ sp \models PairVarUnitMEs}{print(sp.variabilityUnit.name),\ \forall me | me \in sp.mes\ \{print(me.name),\ print(getMetaclassName(me))\}} \tag{3.9}$$

$$\frac{sp \in ss.sources,\ sp \models PairVarMEs}{print(sp.variant.name),\ \forall me | me \in sp.mes\ \{print(me.name),\ print(getMetaclassName(me))\}} \tag{3.10}$$

Equation 3.9 and 3.10 show that printing a sources set ($print(ss)$) will print the $sources$ pairs. Similarly, $print(ts)$ is defined as follows:

$$\frac{tp \in ts.targets,\ tp \models PairVarUnitMEs}{print(tp.variabilityUnit.name),\ \forall me | me \in tp.mes\ \{print(me.name),\ print(getMetaclassName(me))\}} \tag{3.11}$$

$$\frac{tp \in ts.targets,\ tp \models PairVarMEs}{print(tp.variant.name),\ \forall me | me \in tp.mes\ \{print(me.name),\ print(getMetaclassName(me))\}} \tag{3.12}$$

Equation 3.11 to 3.12 shows that printing a targets set ($print(ts)$) will print the $targets$ pairs.

Equation 3.9 to Equation 3.12 show that each pair PairVarUnitMEs and PairVarMEs involved in the violation of a rule will be printed. That includes the name and type of each each model element (me) involved in the violation of a constraint rule. The list of model elements is represented by the attribute mes in the metaclass Pair instantiated by PairVarUnitMEs and PairVarMEs. The definition of Equation 3.9 throughout Equation 3.12 employs a common function $getMetaclassName(me)$ that returns the name of the metaclass of the model element received as parameter.

Finally, the meaning of $print(x)$ is defined by Equation 3.13. It shows that given a string ($x \models String$), it will produce a chain of characters send to the standard output flow of the system.

$$[print(x)] = "x"\ \ iff\ \ x \models String \tag{3.13}$$

Evaluation of Consistency Rule Instance. Our objective when evaluating consistency rule instances is that domain constraints (d) meet model-based specification constraints

(mBS), and model-based specification constraints (mBS) meet domain constraints (d). This logic is expressed by Equation 3.14.

$$[d \rightarrow mBS] \wedge [mBS \rightarrow d] \tag{3.14}$$

A satisfiability solver[15] (SAT for short) can find SAT variable truth assignments of the products[16] that satisfy Equation 3.14. We employ Equation 3.15, which is a modified version of Equation 3.14, to find inconsistencies. Equation 3.15 evaluates to FALSE (i.e., the consistency rule instance is inconsistent) when any of the two sides of the disjunction is satisfiable. Therefore, one can first evaluate only the left-hand side of the conjuntion ($\neg[d \rightarrow mBS]$). If the left-hand side is satisfiable, the entire equation is FALSE. Otherwise, if the left-hand side of the conjuntion is not satisfiable (consistent), the right-hand side ($\neg[mBS \rightarrow d]$) has to be evaluated.

$$\neg[d \rightarrow mBS] \vee \neg[mBS \rightarrow d] \tag{3.15}$$

Based on Equation 3.15 it is possible to evaluate if a specific *Consistency Rule Instance* is consistent or not ($[c.consistent] = True | [c.consistent] = False$). Let $vcc \models VCC$, $d = vcc.dCons$, $ss = c.mBSCons.sourcesSet$, $ts = c.mBSCons.targetsSet$, $cris = vcc.criSet$, where sat (means *satisfiable*) and $\overline{sat}$ (means *not satisfiable*). The evaluation of a consistency rule instance $c | c \in cris$ is shown next:

$$[c] = \begin{cases} [Excludes] & if\ c.mBSCons.type \models Excludes \\ [Requires] & if\ c.mBSCons.type \models Requires \end{cases} \tag{3.16}$$

$$[Excludes] = \begin{cases} False & if\ [\neg[d \rightarrow mBS_{Exc}] = sat \vee \neg[mBS_{Exc} \rightarrow d] = sat \\ True & if\ [\neg[d \rightarrow mBS_{Exc}] = \overline{sat} \wedge \neg[mBS_{Exc} \rightarrow d] = \overline{sat} \end{cases} \tag{3.17}$$

$$[Requires] = \begin{cases} False & if\ [\neg[d \rightarrow mBS_{Req}] = sat \vee \neg[mBS_{Req} \rightarrow d]] = sat \\ True & if\ [\neg[d \rightarrow mBS_{Req}] = \overline{sat} \wedge \neg[mBS_{Req} \rightarrow d] = \overline{sat} \end{cases} \tag{3.18}$$

Equations 3.17 and 3.18 show the same structure. In that structure $\neg[mBS \rightarrow d] = sat$ means that domain constraints (d) do not meet the model-based specification constraints

[15] http://www.satisfiability.org/

[16] Each *Variability Unit* (e.g., a *Feature*) is represented by a variable in the SAT solver. A product is represented by selections of *Variable Units* in a *Variability Model Configuration* (e.g., a selection of *Features* in a *Feature Model Configuration*).

(mBS), and $\neg[d \rightarrow mBS] = sat$ means that model-based specification constraints (mBS) do not meet the domain constraints (d). Equations 3.19 and 3.20 show that each type of model-based specification constraint *(mBS)*, Exclude (mBS_{Exc}) and Requires (mBS_{Req}), has a particular evaluation. That evaluation vary according to the type of Pair, PairVarUnitMEs or PairVarMEs. Let be $cris = vcc.criSet$, $ss = c.mBSCons.sourcesSet$, $ts = c.mBSCons.targetsSet$, $sp \in ss.sources$ and $tp \in ts.targets$.

$$[mBS_{Exc}] = \begin{cases} [mBS_{Exc,varUnit}] & if\ sp, tp \models PairVarUnitMEs \\ [mBS_{Exc,var}] & if\ sp, tp \models PairVarMEs \end{cases} \tag{3.19}$$

$$[mBS_{Req}] = \begin{cases} [mBS_{Req,varUnit}] & if\ sp, tp \models PairVarUnitMEs \\ [mBS_{Req,var}] & if\ sp, tp \models PairVarMEs \end{cases} \tag{3.20}$$

Equations 3.21 and 3.22 show that in an instance of a Consistency Rule Instance for one or more source pairs imply that there are no excluding target pairs in the same product configuration. The name of instances of the Variability Unit are used as terms in the expressions.

$$\begin{aligned}[mBS_{Exc,varUnit}] = \forall sp \bigwedge sp.variabilityUnit.name \\ \rightarrow \neg(\forall tp \bigvee tp.variabilityUnit.name)\end{aligned} \tag{3.21}$$

$$\begin{aligned}[mBS_{Exc,var}] = \forall sp \bigwedge sp.variant.ex \\ \rightarrow \neg(\forall tp \bigvee tp.variant.ex)\end{aligned} \tag{3.22}$$

Equations 3.23 and 3.24 show that in one instance of a consistency rule for one or more source pairs imply that there are any required target pairs in the same product configuration. The name of instances of the Variability Unit are used as terms in the expressions.

$$\begin{aligned}[mBS_{Req,varUnit}] = \forall sp \bigwedge sp.variabilityUnit.name \\ \rightarrow \forall tp \bigvee tp.variabilityUnit.name\end{aligned} \tag{3.23}$$

$$\begin{aligned}[mBS_{Req,var}] = \forall sp \bigwedge sp.variant.ex \\ \rightarrow \forall tp \bigvee tp.variant.ex\end{aligned} \tag{3.24}$$

Obtaining Domain Constraints (d). This subsection addresses how to obtain domain constraints (d) which are mentioned in several equations (e.g., Equations 3.17 and 3.18). Domain Constraints are obtained from the SPL Variability Model. Therefore, they are the same for all possible product configurations. The method to obtain domain constraints depends on the specific Variability Model. This section describes how to obtain domain constraints from the Feature Model employing a translation table between feature model

elements and propositional formulas with its corresponding conjunctive normal form (CNF[17]) which is the standard input form for satisfiability solvers.

Figure 3.21 shows the mapping between feature model elements and propositional logic obtained from the following steps [22]:

1. Each feature maps to a variable of the propositional formula,

2. Each relationship (e.g., requires, optional) is mapped onto one or more formulas,

3. The resulting formula is the conjunction of all the resulting formulas of Step 2 plus an additional constraint assigning TRUE to the variable that represents the root feature, (e.g $SmartHome \longleftrightarrow TRUE$).

The right-most column of Figure 3.21 is the translation to CNF form that will be used by the SAT solver[18]. The mapping rules from propositional logic to CNF form are based on classical logical equivalences [28] such as the double negative law, De Morgan's laws, and the distributive law.

3.6 Tool Support

DCC4SPL tool[19] supports the approach described in this chapter. Figure 3.22 shows a sketch of the high-level architectural view of our tool support identifying two major parts: (1) External tools, and (2) DCC4SPL tool, whose internal components are organized in two subsystems, VCC and VML4RE.

3.6.1 DCC4SPL External Tools

External tools refers to third-party software components developed to be either freely distributed or sold. DCC4SPL employs external tools such as specialized libraries (SPLOT parser and EMF), web applications (SPLOT Editor), and modelling editing platforms for designing and constructing software systems (Ecore-based model editors). They are presented next:

[17] A formula is in CNF form when it is a conjunction of clauses and each clause is a disjunction of literals. CNF formulas have special characteristics: i) a literal and its complement cannot appear in the same clause, ii) the only connectives are: AND ($\wedge$), OR ($\vee$), and NOT($\neg$), and iii) NOT($\neg$) can only be used as part of a literal.

[18] The characters representing the logical operators may change according to the specific SAT solver tool used.

[19] A prototype and more detailed description can be found at:
http://www.mauricioalferez.com/dcc4spl/main.htm

Relationship	Propositional Logic	Propositional Logic in CNF Format
Mandatory	$f_1 \leftrightarrow f_2$	$\left(\neg f_2 \vee f_1\right) \wedge \left(\neg f_1 \vee f_2\right)$
Optional	$f_2 \rightarrow f_1$	$\neg f_2 \vee f_1$
Or	$f_p \leftrightarrow \left(f_1 \vee f_2 \vee \ldots \vee f_n\right)$	$\left(\bigvee_{i=1}^{n}\left(\neg f_i \vee f_p\right)\right) \wedge \left(\neg f_p \vee \bigvee_{i=1}^{n} f_i\right)$
XOr	$\left(f_1 \leftrightarrow \left(\neg f_2 \wedge \ldots \wedge \neg f_n \wedge f_p\right)\right) \wedge$ $\left(f_2 \leftrightarrow \left(\neg f_1 \wedge \ldots \wedge \neg f_n \wedge f_p\right)\right) \wedge$ $\left(f_n \leftrightarrow \left(\neg f_1 \wedge \neg f_2 \wedge \ldots \wedge \neg f_{n-1} \wedge f_p\right)\right)$	$\left(\bigwedge_{i=1}^{n}\left(\neg f_i \vee f_p\right)\right) \wedge \left(\neg f_p \vee \bigvee_{i=1}^{n} f_i\right) \wedge \left(\bigwedge_{i=1}^{a=n-i}\left(\neg f_i \vee \bigvee_{j=i+1}^{b=n} \neg f_{j+1}\right)\right)$
Requires	$f_a \rightarrow f_b$	$\neg f_a \vee f_b$
Excludes	$\neg\left(f_a \wedge f_b\right)$	$\neg f_a \vee \neg f_b$

Figure 3.21: Mapping feature model elements to propositional logic and CNF ([22][28]).

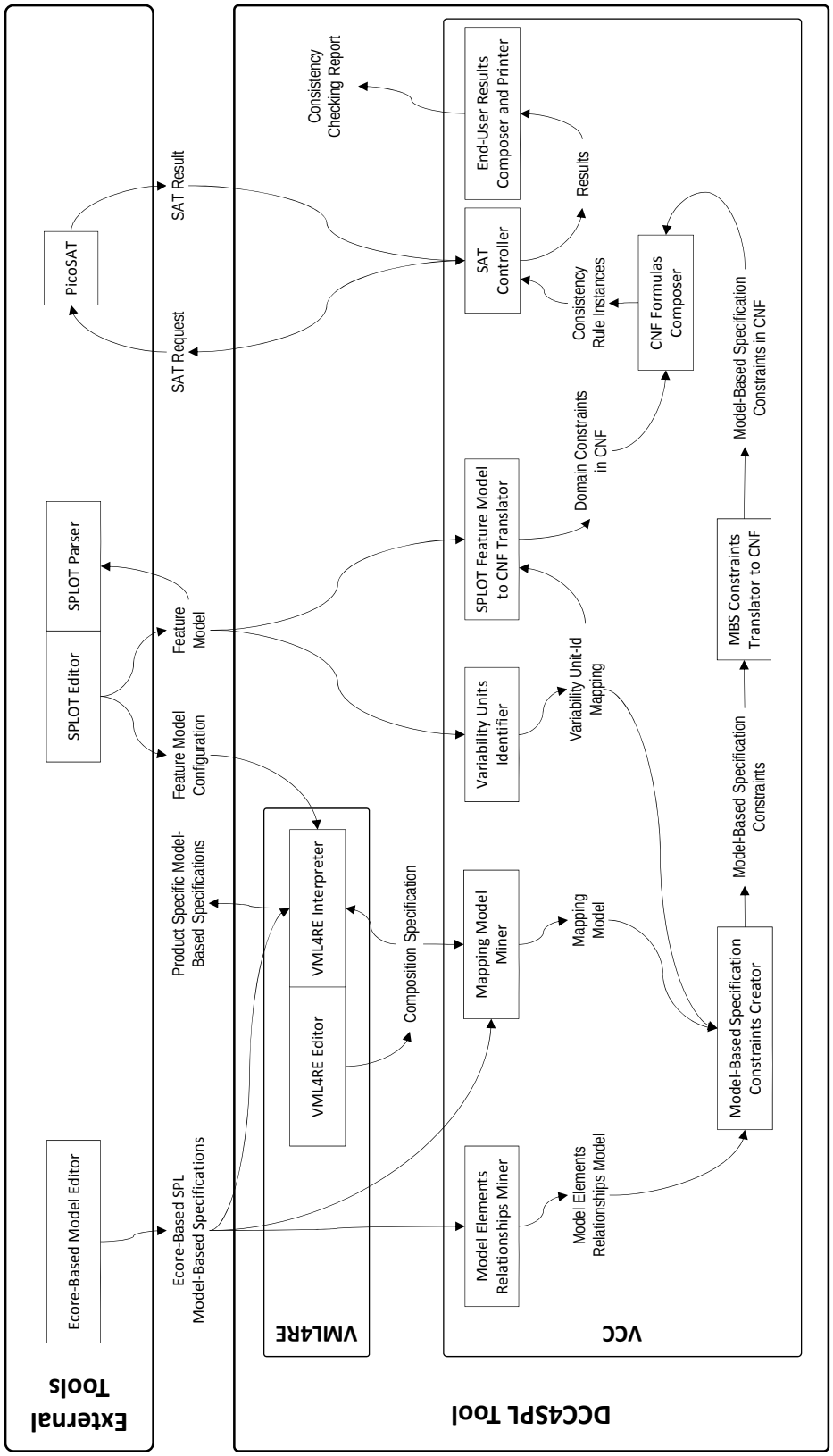


Figure 3.22: DCC4SPL tool support high-level architectural view.

Ecore-Based Model Editor. SPL model-based specifications can be written in any Ecore-based[20] modelling tool. Currently, we used Papyrus[21] and Topcased[22] open-source EMF-based editors to create models (e.g., use cases, activity and component diagrams). Developers can also use commercial tools such as Magic Draw[23], Enterprise Architect [24] or IBM Rational Rose[25] that can export their metamodels to Ecore.

SPLOT Editor and Parser. SPLOT[26] allows us to share and edit our feature models and feature model configurations collaboratively via web. It generates an initial CNF formula of the feature models and its constraints that DCC4SPL translates to the CNF format understood by SAT solvers (It is possible to use other feature model editors that translate to CNF or to implement the translation based on the mapping patterns described in Figure 3.21).

PicoSAT. We employed PicoSat[27] to determine the satisfiability of each formula. We chose PicoSAT because it is fast, its implementation is very compact (it ranges from 91 KB to 171 KB according to the platform), and we had the collaboration of an expert in SAT solvers that helped us to use it appropriately.

3.6.2 DCC4SPL Tool

The main parts of the DCC4SPL tool are two interrelated subsystems (VML4RE and VCC). The internal components in those subsystems create or manipulate models, and communicate with external tools. They are deployed as Eclipse plug-ins encoded in the Java language. The main internal components of the DCC4SPL tool are presented next:

VML4RE Editor. VML4RE language uses EMFTEXT[28] which provides the software infrastructure to derive a concrete syntax and plug-ins based on the metamodel of the language written in Ecore. EMFText separates concrete syntax from abstract syntax, easing maintenance of the languages. The concrete syntax chosen for VML4RE is very close to the HUTN (Human Usable Notation)[29] provided by EMFTEXT.

VML4RE Interpreter. The composition specifications created using the VML4RE editor must be understood to actually perform the composition of the model-based specifications. Therefore, the interpreter first parses the composition specification file and creates an

[20] www.eclipse.org/emf/
[21] www.eclipse.org/papyrus/
[22] http://www.topcased.org/
[23] https://www.magicdraw.com/
[24] http://www.sparxsystems.eu/enterprisearchitect/
[25] http://www-01.ibm.com/software/awdtools/developer/rose/
[26] http://www.splot-research.org/: Software Product Line Online Tools
[27] http://fmv.jku.at/picosat/: PicoSAT: Pico satisfiability solver. We thanks to Alexander Nöhrer for its Java interface for PicoSAT.
[28] http://www.emftext.org/: EMF textual concrete syntax mapper.
[29] http://www.omg.org/spec/HUTN/: The HUTN specification.

internal representation of the model. Next, using the EMF Ecore API and according to the feature model configuration, the interpreter decides which actions to execute. The composition is performed using the Eclipse UML API[30] that already has a set of methods to transform UML models. We are currently developing an alternative method for the transformation of the model-based specifications based on DSL Trans [68, 20]. DSL Trans is a model transformation tool that helps to describe in detail the step by step transformation related to each action and also guarantess that every model transformation terminates.

Model Elements Relationships Miner. We have two implementations of this component. For requirements models, such as use cases and activity diagrams, we obtain an initial list of required and provided elements based on the composition specification and model-based specifications that realize or design the SPL features. For architectural models, we parse the component diagram to obtain the set of provided and required interfaces of each component.

Mapping Model Miner. Parses the composition specification to find the variants and the model elements that each one introduces into the use cases, activity and component diagrams.

Variability Units Identifier. Assigns and persists identifiers for each variability unit. Identifiers are used as variable names when creating constraints in CNF format and during satisfiability checking.

Model-Based Specification Constraints Creator and Translator to CNF. Given the dependencies between model elements (i.e., a Model Elements Relationship Model) and their relationships with variants (i.e., a Mapping Model), it creates the model-based specifications constrains as described in Chapter 14. All the constraints are translated to CNF, a format readable by SAT solvers.

Feature Model CNF Translator. Translates the clauses generated by SPLOT to an appropriate CNF form readable by SAT solvers.

CNF Formula Composer. Generates the formulas that will be passed to the SAT solver, based on the constraints created by the Feature Model CNF Translator and Constraints Creator and CNF Translator components.

3.7 Summary

This chapter described DCC4SPL (Derivation and Consistency Checking of models in early SPLE). It started with an overview of DCC4SPL and an illustrative example from

[30] http://www.eclipse.org/modeling/mdt/?project=uml2

the domain of home automation. Next, it presented the two main parts of DCC4SPL, the Variability Modelling Language for Requirements (VML4RE) and the Variability Consistency Checker (VCC). VML4RE supports the creation of composition specifications and derivation of early models for products in Software Product Line Engineering. VCC supports consistency checking between variability models and model-based specifications.

DCC4SPL was described based on its abstract syntax and semantics, and for VML4RE its representation using a syntactic mapping to a concrete syntax. The abstract syntax of DCC4SPL was presented using a metamodel. This chapter specifies the abstract syntax of VML4RE and VCC employing a metamodel. This metamodel considers the concepts (metaclasses) this language deals with, how they interrelate (using containment and non-containment references) and the properties (attributes) they have. The semantics of DCC4SPL and its two main parts, VML4RE and VCC, describes what happens when their instances are interpreted by their tool support. The semantics harmonizes and complements with operational semantics the informal descriptions and definitions provided in previous chapters. The concrete syntax of VML4RE is an exemplar textual syntax specified by means of a grammar and related to the abstract syntax of the language. Finally, this chapter presents the tool support created for VML4RE and VCC based on the abstract and concrete syntaxes and their established semantics.

4

Validation

This chapter provides an overview of the validation of DCC4SPL which aims to better understand how and to what extend it guarantees effective consistency checking and derivation of product-specific models in early SPLE. We base our validation on different methods ranging from experimentation with case studies, comparative metrics, prototypes, and feedback obtained during the elaboration and presentation of peer-reviewed scientific publications. This chapter presents the goal, questions, attributes and metrics that guided our validation, summarizes the case studies used in the research papers, reviews the questions raised by each research topic and the methods used to answer them.

4.1 Goal and Research Questions

There are some base elements that we employed to define the goal: the object and purpose of the study, the attributes that we focused on, the point of view of the people involved in the study, and the context in which the study took place. Taking into account those elements, the goal is defined as follows:

To analyse DCC4SPL (the **object**) to understand how and to what extend *it guarantees effective consistency checking and derivation of product-specific models early in SPLE* (the **purpose**),

with respect to the **attributes** of *modularity* of the features specifications (how localized is each feature specification along the models), *stability* (how changes in the specifications are required to evolve an SPL from one configuration to another), *expressiveness* (how verbose are the specifications addressing those changes), *vocabulary familiarity* (whether or not the compositions specifications use a vocabulary familiar to the developer), *derivation flexibility* (whether or not it supports different variability composition mechanisms),

Question 1. How to express model derivation at the level of abstraction that the developers require?
- Question 1.1. Modularity: Which techniques (and to what extent) support improved modularization of feature specifications?
- Question 1.2. Stability: Which techniques (and to what extent) support stability of specifications (i.e., early models, composition and configuration knowledge) after SPL evolution?
- Question 1.3. Expressiveness: Which techniques (and to what extent) benefit expressiveness of specifications after SPL evolution?
- Question 1.4. Vocabulary familiarity: How to facilitate to developers to write composition specifications in a language familiar to them?
- Question 1.5. Derivation flexibility: How to support positive and negative variability composition mechanisms?
Question 2. How to support consistency between the features model and other models?
- Question 2.1. Consistency checking genericity: How to perform consistency checking independently from the languages specificities.
- Question 2.2. Consistency checking multi-view awareness: How to check consistency between multiple models and the variability specification independently from the number of languages employed.
- Question 2.3. Consistency checking scalability: How to check consistency of a growing number of products?

Table 4.1: Main research questions related to the research topics and the sub-questions used to address them.

consistency checking genericity (whether or not it can verify consistency independently of languages specificities), *consistency checking multi-view awareness* (whether or not it can verify consistency between multiple models and the variability specification), and *consistency checking scalability* (whether or not it can verify consistency of a growing number of products),

from **the point of view of** software engineers

in the **context** of three case studies proposed by the SPLE research community and industrial partners in the European AMPLE project.

Taking into account this goal in our validation, Table 4.1 shows the two main questions related to the research topics (Section 1.2) as well as some sub-questions that will be mapped to attributes and metrics in Subsection 4.2.

4.2 Attributes and Metrics

This section describes the attributes and their corresponding metrics considered in DCC4SPL validation. The attributes, related to the questions in Table 4.1, can be organized in two groups according to the type of metric value: (1) Attributes with quantitative metric values (Section 4.2.1), and (2) Attributes with qualitative metric values (Section 4.2.2).

Attribute	Metric
Modularity	Degree of scattering of features [37].
	Degree of focus of models [37].
Stability of the specifications	Number of steps introduced or changed between two releases [26].
Stability of the compositions	Number of compositions items introduced or changed between two releases [26].
Stability of the configuration knowledge	Number of configuration items introduced or changed between two releases.
Expressiveness of the composition	The ratio between the number of matched join points and the number of composition items [26].
Expressiveness of the configuration knowledge	Number of tokens required to specify the configuration knowledge.

Table 4.2: Attributes used during quantitative validation.

4.2.1 Attributes with Quantitative Metric Values

This subsection presents the attributes whose metric values are quantities or else quantifiable data. Table 4.2 summarizes the attributes and related metrics.

Modularity. Our modularity investigation relies on two metrics customized from [37] and [24]: Degree of Scattering of Features (DoS) and Degree of Focus of Models (DoF). According to Equations 4.1 and 4.2, DoS quantifies the concentration of a feature[1] over each model $m \in M$ (the set of models). Values of DoS are normalized between 0 (completely localized) and 1 (completely scattered). The greater the DoS of a feature f is, the greater is the probability of reviewing different models when the specification of f has to evolve. In Equation 4.1, $\mid M \mid$ denotes the cardinality of the set M.

$$DoS(f) = 1 - \frac{\mid M \mid \sum_{m \in M}(Concentration(f,m) - \frac{1}{\mid M \mid})^2}{\mid M \mid -1} \quad (4.1)$$

$$Concentration(f, m) = \frac{number\ of\ model\ elements\ in\ m\ assigned\ to\ f}{number\ of\ model\ elements\ assigned\ to\ f} \quad (4.2)$$

Likewise, according to Equation 4.3 and 4.5, DoT considers how many model elements of a model are related to each feature $f \in F$ (the set of features). Values of DoT are similarly normalized between 0 (completely focused) and 1 (completely tangled). The greater the DoT of a model m is, the greater the probability of reviewing m when one of the related features changes. Usually we use the metric Degree of Focus (DoF) to refer to low tangling (DoT near to 0) in models. DoF in Equation 4.4 is easily derived from DoT and corresponds to $1 - DoT$. Thus, the lower the DoF of a model m is, the higher is the tangling (DoT) of features it specifies.

[1] These metrics can be interpolated to other modularization techniques. Thus, a *feature* may be replaced by the more generic term *Variability Unit*.

High values in the degree of focus and low values in the degree of scattering are usually associated to well-modularized systems [37, 64, 65].

$$DoT(m) = 1 - \frac{\mid F \mid \sum_{f \in F}(Dedication(m,f) - \frac{1}{|F|})^2}{\mid F \mid -1} \tag{4.3}$$

$$DoF(m) = 1 - DoT(m) \tag{4.4}$$

$$Dedication(m,\ f) = \frac{number\ of\ model\ elements\ in\ m\ assigned\ to\ f}{number\ of\ model\ elements\ of\ m} \tag{4.5}$$

Note that to evaluate the metrics in equations 4.2 and 4.5 we have to assign features to the individual model elements of a model specification. We follow the *configuration dependency analysis* as a guide [24], considering that a model element me depends on a feature f if, and only if, the selection of f triggers the configuration (transformation) of me.

Stability. Regarding our stability assessment, we adapted a metrics suite that has also been validated and used to compare approaches for requirements engineering modelled using advanced separation of concerns techniques [26]. These metrics originally quantify the stability of specifications and code elements that represent a software artefact in the context of evolutionary scenarios. In our study, we used them to quantify the stability of early models along the different releases of an SPL. It is possible to measure the stability of the specifications and the stability of the compositions. According to Equations 4.6 and 4.7, they are quantified by the number of modified or introduced specification items (i.e., model elements and models), and the number of modified or introduced composition items (e.g., pointcuts in advices and actions) between two subsequent releases. In addition, we have also proposed a metric in Equation 4.8 that quantifies the stability of the configuration knowledge in terms of number of modified or introduced configuration items (e.g., annotations and expressions).

$$Stability\ of\ specifications = \frac{number\ of\ added\ and\ changed}{specification\ items} \tag{4.6}$$

$$Stability\ of\ compositions = \frac{number\ of\ added\ and\ changed}{compositions\ items} \tag{4.7}$$

$$Stability\ of\ configuration\ knowledge = \frac{number\ of\ added\ and\ changed}{configuration\ items} \tag{4.8}$$

Attribute	Description
Vocabulary familiarity.	Whether or not the compositions specifications use a vocabulary familiar to the modeller.
Derivation flexibility.	Whether or not it supports positive and negative variability compositions mechanisms.
Consistency checking genericity.	Whether or not it can abstract from languages specificities
Consistency checking multi-view awareness.	Whether or not it can verify consistency between multiple models and the variability specification independently of the number of languages employed.
Consistency checking scalability.	Whether or not it can check consistency of a growing number of products.

Table 4.3: Attributes used during qualitative validation.

Expressiveness. For measuring the expressiveness of the configuration knowledge, we count how many tokens are required to map features (i.e., Variability Units) to other models. Although the unit to quantify expressiveness of the configuration knowledge is rather low level, it allows us to uniformly assess the different representations of the configuration knowledge. To measure expressiveness of the compositions, we use the notion of reachability [26]. According to Equation 4.9, reachability is calculated as the ratio between the number of matched join points (model fragments in model-based specifications) and the number of composition items.

$$Reachability = \frac{number\,of\,matched\,join\,points}{number\,of\,composition\,items} \tag{4.9}$$

4.2.2 Attributes with Qualitative Metric Values

This subection presents the attributes whose metric value answers “whether or not DCC4SPL satisfies a property”, thus, they can be compared to a binary variable with two possible values "Yes" and "Not". Table 4.3 summarizes these attributes.

Vocabulary Familiarity. Every language has a concrete syntax which is composed of different symbols that integrate the vocabulary of the language. To evaluate vocabulary familiarity we determine whether or not the compositions specifications use a vocabulary familiar to the modeller. We consider that a modeller is familiar with the vocabulary of the language used to write the composition specifications when s/he has the knowledge and awareness of this vocabulary gained by personal experience. Therefore, we consider that an approach provides a vocabulary familiar to the modeller when it employs the name of model elements used in the concrete syntax instead of names and operators that refer to the abstract syntax of the composition specification language.

Derivation Flexibility. The lack of flexibility to choose the derivation mechanism is related to the problem of tangling between configuration knowledge and model-based

specifications. Mixing configuration knowledge (e.g., annotations or stereotypes with the names of features) in the model-based specifications allows only one kind of composition, generally *negative variability composition*.

In negative variability composition there is an initial model that is transformed to obtain particular models for other products with less features. The transformation in performed removing or replacing model elements related through annotations or stereotypes with variable features that are not selected in the feature model.

However, to have an initial monolithic model with all kinds of variabilities modelled since the very beginning is not always possible and it does not necessarily facilitates to understand the models. Also, it can lead to ambiguity when other annotations (e.g., stereotypes) are used in the models, as usually happens in domain specific languages or models created based on profiles. Nevertheless, sometimes when the SPL model-specifications are small, it can be feasible to design all the model fragments related to the features in one or few initial models.
Therefore, we consider that an approach provides derivation flexibility when, in addition to negative variability composition, it also provides *positive variability composition*. Positive variability composition means to start with a small core model whose parts are related to mandatory features. The core model is suitable to be transformed (e.g., adding model elements related to variable features) to obtain particular models for products that include variable features. Also, a combination between possitive and negative variability composition should be possible; for example, when developers want to model a core model together with some model fragments related to the most frequently selected variable features.

Consistency Checking Genericity. Genericity means that the approach for consistency checking should be specified independently of the languages used to model variability and its related model-based specifications. The concrete consistency checking rules are then instantiated when needed for specific languages.

Therefore, we consider that an approach supports consistency checking genericity when it can abstract from concrete languages and formalises the consistency checking approach as concepts. That formalization should be expressed independently of the languages used to write model-based specifications and the modularization technique used to decompose them. Next, when it is necessary, those concepts can be instantiated to concrete concepts of particular languages to evaluate whether or not the SPL specifications are consistent with the variability specification.

Consistency Checking Multi-View Modelling Awareness. When model-based specifications are used to represent early models (e.g., in form of use cases, activity or component diagrams) and feature models, consistency goes beyond syntactical or semantic errors of each kind of model in isolation. For example, an actor that is not associated with any use case, a dangling node in a component diagram, a loop without exit conditions in activity

diagrams or specific set of features that are both simultaneously (and incorrectly) declared as excluding and depending. It means that we aim at taking into account constraints that are not merely expressed in terms of only one language's metamodel which is generally well supported by UML editors (e.g., using OCL or hard-coded restrictions particular of each editor) or feature model editors (e.g., using domain constraints expressing features interdependencies, and hard-coded restrictions that constrain the construction of the models to conform to their metamodel).

Much of consistency checking difficulty lies on maintaining consistency among several, interrelated models. This can become a time-consuming and error prone task given that the number of ways to compose feature realizations grows exponentially with the possible number of SPL features that can be used in a particular product.

Consistency checking multi-view awareness is similar to the consistency checking genericity attribute because both aim at making the consistency checking process more independent of particular variability modelling, system modularization and languages particularities. However, multi-view awareness focusses on the support for the creation of consistency rules that involve several views of the system (which may be usually modelled using different languages). Therefore, we consider an approach to be multi-view aware if it can check consistency between multiple models and the variability specification independently of the number of languages employed to write them.

Consistency Checking Scalability. One of the main characteristics of SPLs is the possible large number of different valid combinations of product features and their relationships with model fragments. Thus, to apply consistency checking product by product is feasible only for small SPLs [17]. It is crucial that consistency checking checks the entire SPL specification against the variability model to guarantee that any possible selection of variability units (e.g., features) renders a valid product variant.

Therefore, we consider that an approach is scalable in checking consistency, when the time to find inconsistencies does not grow exponentially with the amount of possible numbers of products.

4.3 Case Studies

This subsection briefly presents the three case studies employed to validate DCC4SPL and to answer the research questions. These case studies cover different application domains, such as home automation, mobile applications and emergency systems. Each one comes from a different source (e.g., research project technical reports, dissertations and scientific journal papers):

- *Smart Home* was defined as a case study in the AMPLE project [71]. The Smart Home comprises significant aspects of modern home automation domain such as security, HVAC (Heating, Ventilating, and Air Conditioning), illumination control, remote

control and multiple user interfaces. Smart Home was discussed in technical reports [10, 5] and all the papers included in the Part II of this dissertation.

- *Car Crash Crisis Management System (CCCMS)* was proposed by a group of international researchers as a common case study to compare different Aspect-Oriented Modelling (AOM) approaches [56, 55]. The CCCMS comprises the aspects needed to facilitate the process of identifying, assessing, and handling a crisis situation by orchestrating the communication between all parties involved in handling the crisis, allocating resources and providing information to determined users. CCCMS was used in Chapter 12.
- *Mobile Photo* was developed in the University of British Columbia [95, 43]. Mobile Photo (SPL) comprises the aspects needed to manipulate photos on mobile devices. Mobile Photo was used in Chapter 14.

These case studies were selected because:

1. They are of different application domains while still understood by readers in general given their familiarity with the domains.
2. We had previous experience in modelling variability and part of their model-based specifications. The use scenarios of Smart Home were inspired on the requirements of the system set by one of our industrial partners in the European AMPLE project [71]. Mobile Photo has also been used in the context of AMPLE.
3. The Car Crash Crisis Management System case study is a benchmark in AOM and their models are described in [54].

4.4 Validation Settings

This subsection presents the settings of the DCC4SPL validation employing the three case studies just introduced. In particular it presents the phases and assessment procedures of our study, taking into account the metrics suite adopted (Subsection 4.2).

4.4.1 Study Phases and Assessment Procedures

Our study was organized in five major phases:

1. Specification of the CCCMS using four alternative approaches.
2. Specification of evolution scenarios for all the approaches.
3. Quantitative and quantitative assessment of the different specifications and releases of the CCCMS.
4. Specification of the Smart Home and Mobile Photo in addition to the specification of CCCMS.

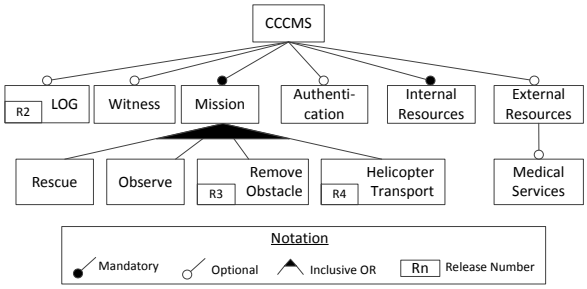


Figure 4.1: Feature model for the CCCMS SPL.

5. Qualitative and quantitative assessment of the specifications and consistency checking attributes of the Smart Home, Mobile Photo and the rest of the assessment of CCCMS (part of the assessment of CCCM is already done in Phase 3).

All specifications were written according to alignment rules, which were necessary not only to verify that good practices were used in the approaches, but also to ensure that the comparison of the specifications was equitable and fair. Three researchers performed these alignment activities. All misalignments found were discussed between the study participants and eventual corrections were applied to the specifications to guarantee their alignment. For example, we ensured that: (1) every variability was modularized using the appropriate modularization and composition mechanisms of each approach; (2) textual and graphic-based approaches used an equal number of elements that represent the same abstraction (such as activities and textual steps for the scenarios); and (3) the specifications reflect the same functionalities/features.

Phase 1. CCCMS was specified using four different modularization and composition mechanisms from four approaches: PLUSS [41, 42], Model Templates (MTs) [30], MSVCM [24] (see the description of the approaches and details of the evaluation in Chapter 12).

Phase 2. From the models detailed in [56] we developed a set of incremental releases for the CCCMS (they are available online[2]). Considering the feature model shown in Figure 4.1, we have defined a base release (R1), consisting of the features Authentication System, Rescue Mission, Witness, Medical Services, Internal and External Resources and Observe Mission. After that, the specifications were evolved to address the change scenarios corresponding to the releases R2—R4, that appear in Figure 4.1. The features inserted in the releases required the introduction of new scenarios and changes to existing ones. These change scenarios allowed us to exercise the different modularization and composition mechanisms provided by each approach, to observe their modularity, stability, and expressiveness.

[2]http://www.mauricioalferez.com/dcc4spl/main.htm

Phase 3. This phase employed our metrics suite (see Section 4.2) to analyze and compare the obtained results for the different specifications chosen in Phase 1.

Phase 4. In this phase, our objective was to determine if DCC4SPL can find inconsistencies between variability units and different kinds of model-based specifications. We defined feature models and model-based specifications for each case study, and the mapping between model elements and variants using VML4RE and lightVC[3] (they are available online[4]). Table 4.4 summarizes the rules that we used in this evaluation. We used 7 rules of type Requires and 4 of type Excludes, considering 4 kinds of relationships between model elements:

1. Rules that consider INCLUDES relationships (Rules 1, 8): an includes relationship, in which one use case (the base use case) includes the functionality of another use case (the inclusion use case), supports the reuse of functionality in use-case diagrams. In DCC4SPL this kind of rule suggests including in a product the inclusion model element if the base model element is included.

2. Rules that consider CONTAINMENT relationships (Rules 2, 3, 6, 9, 11): the contained model element is the one that requires a container. In UML a common container is the PACKAGE, but others exist, such as ACTIVITY that contains model elements represented in activity diagrams. In DCC4SPL this kind of rule suggests including in a product the container model element if the contained model element is included.

3. Rules that consider the USAGE and REALIZATION relationships (Rules 7, 10): USAGE relationship is a type of dependency relationship in which one model element (the client) requires another model element (the supplier) for full implementation or operation. In this experiment we consider USAGE and INTERFACE REALIZATION relationships in component diagrams employing components as clients, and interfaces as suppliers. In DCC4SPL this kind of rule suggests including in a product the supplier model element if the client model element is included.

4. Rules that consider the GENERALIZATION relationships (Rules 4-5): a generalization relationship is a relationship in which one model element (the child) is based on another model element (the parent). Generalization relationships are used in class, component, deployment, and use case diagrams to indicate that the child receives all of the attributes, operations, and relationships that are defined in the parent. In our experiment we considered generalization relationships between ACTORS and between USE CASES. In DCC4SPL this kind of rule suggests including in a product the parent model element if the child model element is included.

[3]This is a version of VML with actions used to remove or add components, and connect them through their interfaces.

[4]http://www.mauricioalferez.com/dcc4spl/main.htm

Rule #	Description	Relationship	SH	CC	MP
1	Required *inclusion Use Case* when *base Use Case* included	Inclusion	*	*	
2	Required *Package* when any of its *Use Cases* are included	Containment	*	*	
3	Req. *Package* when any of its contained *Packages* is included	Containment	*	*	
4	Req. *Use Case* when any of its *children Use Cases* are included	Generalization	*	*	
5	Req. *Actor* when at least one of its *children Actors* are included	Generalization	*	*	
6	Req. *Activity* if at least one of its *Opaque Actions* are included	Containment	*	*	
7	Required *required Interfaces* when *Component* is included	Usage/Realization	*		*
8	Excluded *base Use Case* when *inclusion Use Case* is excluded	Inclusion	*	*	
9	Excluded *Use Case* when its *Package* is excluded	Containment	*	*	
10	Excluded *provided Interfaces* when its *Component* is excluded	Usage/Realization	*	*	*
11	Excluded *Opaque Actions* when its *Activity* is excluded	Containment	*	*	

Table 4.4: Summary of the rules implemented in our study and applied (*) for Smart Home (SH), CCCMS (CC), Mobile Photo (MP).

Phase 5. Finally, in Phase 5 we evaluated the metrics related to consistency checking from our metrics suite (see Section 4.2) and next we analysed the results.

4.5 Quantitative and Qualitative Validation

This subsection overviews the results shown in research papers in Part II and it is divided in two parts: qualitative validation and quantitative validation. Each part relates the validation results with the attributes and metrics described in Subsection 4.2 - Attributes and Metrics.

4.5.1 Qualitative Validation

Vocabulary Familiarity. It is addressed by DCC4SPL as it requires the developers to use the name of the symbols in the concrete syntax of the early models in the *Actions*, instead of using directly the abstract syntax to express model transformations. Figure 3.18 in Chapter 3 presented some examples of how easy it is to express actions in DCC4SPL. One of those examples was about the association of a use case with name *useCaseX* with an actor named *actorY*. That association can be expressed in DCC4SPL via VML4RE as: "ASSOCIATE USECASE : "*useCaseX*" TO ACTOR : "*actorY*". To obtain the same results programming that action directly in a transformation language would require the developer to know the name of the metaclasses and their relationships (i.e., the abstract syntax of the language). If the developer was to implement such association, s/he would have to understand the UML metaclassses Property, Association, UseCase, Actor, their relationships, as well as their attributes (e.g., "Aggregation kind" in Association).

Derivation Flexibility. It is addressed by DCC4SPL as it supports positive, negative and a combination between positive and negative variability composition. Figure 3.13 shows that it has four kinds of actions, Insert, Connect, Replace and Remove. Insert and

Connect actions support positive variability composition, as they add model elements into a target model. Remove action supports negative variability composition because it takes out model elements from a target model. Replace and all the others actions support a combination of both negative and positive variability composition since they allow, in the same composition process and composition specification, to remove and add model elements from/to a target model.

Consistency Checking Genericity. It is addressed by DCC4SPL as it formalises the consistency checking approach with concepts expressed as metaclasses and relationships in the abstract syntax of VCC. Then, DCC4SPL specifies and performs consistency checking independently of the specialization of the abstract metaclasses Variability Unit, Variability Model, Model-Based Specification and Model Element. DCC4SPL also provides two types of consistency checking according to the kind of modularization employed in the SPL system. The first is based on Variability Units; the second is based on Variants[5]. Variability Units and Variants in DCC4SPL allows using it with, for example (1) Feature-Oriented Software Development approaches which employ one specialization of Variability Unit called Feature to identify Feature Modules that contain only the model elements related to that feature, and (2) Model Templates and VML approaches that employ *Expressions*, usually Feature Expressions, that can be associated to any model element contained in any kind of system module.

Consistency Checking Multi-View Awareness. It is addressed by DCC4SPL as it takes into account constraints that are not merely expressed in terms of only one language's metamodel. With DCC4SPL, developers can check consistency between multiple models and the variability specification, even if they are written in different languages. This is supported in VCC where each instance of Model-Based Specification Constraint defines sets of model elements (i.e., instances of SourcePairSet) that depend or are incompatible with other sets of model elements (i.e., instances of TargetPairSet). Each model element can be associated to a different instance of Model-Based Specification that may have its own language metamodel and designs one or several views of the system.

Consistency Checking Scalability. It is addressed by DCC4SPL because:

1. DCC4SPL does not check each product separately (i.e., verification during SPL application engineering). Instead, it checks consistency for the entire SPL specification against the variability model to guarantee that any possible selection of variability units renders a valid product variant (i.e., verification during SPL domain engineering). The result is that the time to find inconsistencies does not grow exponentially with the amount of possible number of products. However, according to Apel [17]

[5] According to the DCC4SPL metamodel each Variant is associated to an Expression. Each Expression is composed of Variability Units and logical operators.

Case Studies	Smart Home	CCCMS	Mobile Photo
Features	60	12	14
Main model elements in early models	36 Use Cases, 14 Packages, 12 Activity Diagrams, 59 Activities.	11 Activity Diagrams, 70 Activities.	12 Components, 15 Interfaces.
Valid product variants	One billion	240	16

Table 4.5: Information of the size in the case studies.

consistency checking performed on a product by product basis (i.e., verification during application engineering) is still feasible for small SPLs.

2. DCC4SPL transforms the variability model and consistency rule instances to CNF clauses. Therefore, the performance of our approach related to obtain the satisfiability value of a CNF formula is directly proportional to the efficiency of the SAT solvers. State-of-the-art SAT solvers, such as PicoSAT[6], are able to handle large number of clauses and are used in industrial applications[7].

These two points about scalability are complemented with Table 4.5, which provides information on the case studies such as the number of features, the number main model elements, and the number of product variants that can be produced. Smart Home has 60 features and concentrates on use case and activity diagrams, CCCMS has 12 features and concentrates on activity diagrams, and Mobile Photo has 14 features and focuses on component diagrams. According to the SPLOT feature model analyser, the Smart Home feature model allows the generation of one billion product variants, 240 for CCCMS and 16 are possible for Mobile Photo.

Table 4.6 also provides data on the size of the case studies with respect to VML4RE composition specifications, the number of consistency rules implemented in VCC, the number of consistency rule instances created automatically when checking consistency, and the time taken to check consistency and present results to the developers. The VML4RE composition specification in Smart Home required a larger number of constructs of type Variant and Action (28 variants and 108 actions) in comparison with CCCMS (7 variants and 17 actions) and Mobile Photo (7 variants and 29 actions). However, the time to check consistency in the three case studies did not exceed one second when run on an Intel Core-Duo i5 at 2.4 Ghz. From the rules presented in Table 4.4, Smart Home used 9 while CCCMS and Mobile Photo used 2 each one, according to the kind of model elements in each case study.

The results in consistency checking consist on the description of each consistency rule instance analysed and the particular inconsistent pairs (model elements, variants (including its feature expression)) which require or are incompatible with other pairs.

[6] fmv.jku.at/picosat/
[7] http://www.satlive.org/

Case Studies	Smart Home	CCCMS	Mobile Photo
VML4RE variants	28	7	7
VML4RE actions	108	17	20
Implemented rules	9 (all except rules 7,10)	2 (6,11)	2 (7,10)
Rule instances checked	71		10
Time taken to check consistency	805 miliseconds	755 miliseconds	745 miliseconds

Table 4.6: Number of elements in the composition specifications and consistency checking time in the case studies.

4.5.2 Quantitative Validation

Table 4.7 summarizes the evaluation results provided in Sections "Modularity Assessment" to "Stability Assessment" included in Chapter 12. For each metric we assigned a symbol that helps distinguishing which techniques have good, average or bad results in comparison with the others[8]. The percentages limits used were: bad ≥ 67; average < 67 and ≥ 33; good < 33[9]. For stability and expressivity of the compositions that only applied to two techniques (i.e., the compositional approaches: VML4RE and MSVCM), we used short arrows indicating which one had a better value (upwards arrow) than the other (downwards arrow).

In general, the lower the value obtained for a metric, the better the approach for the correspondent attribute. However, metrics such as Degree of Focus (DoF) and Reachability follow an inverse logic, therefore higher values are interpreted as desired values. For example, in DoS the percentage limits to assign the symbols were: good ≥ 67; average < 67 and ≥ 33; and bad < 33.

Attribute	Metric	PLUSS	MT	MSVCM	VML4RE
Modulartiy	Average DOS	0,113	0,100	0,058	0,000
	Average DOF	2,960	3,310	4,000	4,000
Stability of specifications	Added steps	32	25	21	21
	Impact	4	5	3	3
Stability of composition	Added compositions	-	-	2,75	4,25
Stability of CK	Modifications	2	6	1	2
	Insertions	32	18	2	2
Expresiv. of compositions	Reachability	-	-	1,5	1,0
Expresiveness of CK	Percentual Growth	321,43	109,62	34,21	134,43

Table 4.7: Summary of quantitative validation. The upwards arrow means "Good", the rightwards arrow means "Average", and the downwards arrow means "Bad".

Modularity. Modularity for each technique was measured as the means of DoS and DoF for the four releases. The compositional approaches VML4RE and MSVCM had better

[8]The assignment of each symbol was determined automatically using the conditional formatting feature of MS Excel which assigned symbols to series of values based on percentages.

[9]These limits only follow an equitative division of 100 in three parts.

results in both DoS and DoF. Their scenario specifications better applied a modular design. VML4RE and MSVCM help to specify each feature separately in only one or few scenarios, which leads to DoS values very close to 0 for MSVCM and 0 for VML4RE. Similarly, VML4RE and MSVCM specified each scenario focusing on only one or few features which resulted in a good DOF=4 in comparison with Model Templates (DOF=3,31) and PLUSS (DOF=2,96). We believe that the annotations mechanism used by Model Templates and PLUSS fail to improve modularity of scenario specifications, even with few features that are scattered through the system such as LOG and Authorization.

Stability of the specifications, compositions and configuration knowledge. Stability of specifications, composition and configuration knowledge (CK) was measured as the sum of all the individual values for stability metrics obtained in all the releases. Similarly, the Impact was measured as the sum of added and modified scenarios in all the releases. The compositional approaches VML4RE and MSVCM obtained the same values for the metrics of stability of the specifications, this means that the most noticeable differences between the compositional approaches (apart from their notation) are found in their expressivity and not in the specification of the scenarios or modularity itself. On the other hand, PLUSS was the best technique to keep almost intact CK specifications (Modifications=2) however, it was done at the price of many insertions (Insertions=32). A different phenomenon happens with the rest of the approaches that faced evolution of CK combining few modifications and insertions of CK.

Expressiveness. Expressiveness of CK was measured as the mean of Reachability for the four releases. VML4RE had a low Reachability (=1) compared to MSCVM (=1.5). The Percentual Growth of Expressiveness of CK in MSCVM (=34,21) was the best while in PLUSS it was the worse (=321,43). The results of expressivity of VML4RE are similar to the ones of Model Templates (Percentual Growth of Expressiveness of CK=134.43 and CK=109.62) that does not have any separate configuration knowledge model. We see that the lack or presence of quantification mechanisms affected Expressivity, and Expressivity affected Stability of compositions and CK. For example, the lack of quantification mechanisms in VML4RE limited the Reachability of its pointcuts and influenced negatively the Stability of compositions and CK because of new required variants and compositions items to match elements introduced in new scenarios.

4.6 Summary of Results

Table 4.8 summarizes the results presented of the validation with respect to the attributes and metrics defined in Subsection 4.2. The qualitative attributes receive a “Yes” when DCC4SPL addressed the property while the quantitative attributes receive a “Partial” for partially addressed or “Good” when it is addressed.

Attribute	Value
Vocabulary familiarity	Yes
Derivation flexibility	Yes
Consistency checking genericity	Yes
Consistency checking multi-view awareness	Yes
Consistency checking scalability	Yes
Modularity	Good
Stability of the specifications	Good
Stability of the compositions	Less stable than MSCVM
Stability of the configuration knowledge	Good
Expressiveness of the composition	Less expressive than MSCVM
Expressiveness of the configuration knowledge	Average

Table 4.8: Summary of validation results for DCC4SPL.

4.7 Summary

The validation of DCC4SPL aimed at understanding to better understand how and to what extend it guarantees effective consistency checking and derivation of product-specific models early in SPLE.

This was accomplished in four steps. First, the research question was divided into two main questions and each one into sub questions.

Second, we described the metrics suite employed in the evaluation of 11 attributes: modularity, stability of the specifications, stability of the compositions, stability of the configuration knowledge, expressiveness of the composition, expressiveness of the configuration knowledge, vocabulary familiarity, derivation flexibility, consistency checking genericity, consistency checking multi-view awareness and consistency checking scalability.

Third, we presented the three case studies employed in the evaluation (Smart Home, Mobile Photo and Car Crash Crisis Management System), and the phases followed for the validation. These phases first deal with the validation of different SPL releases of CCCMS, and then, with the other case studies.

Finally, we discussed the results of the validation. These results show that DCC4SPL guarantees effective consistency checking and derivation of product-specific models in early SPLE taken into account nine attributes. However, the two attributes related to expressiveness of the composition and configuration knowledge of DCC4SPL need to be improved as it may benefit the stability of compositions which did not get the best values during the validation presented in this chapter.

5

Conclusions

In this dissertation we addressed the issue of *guaranteeing effective consistency checking and derivation of product-specific models early in SPLE* with a novel and tool-supported framework called DCC4SPL. DCC4SPL is composed of the Variability Modelling Language for Requirements (VML4RE) and the Variability Consistency Checker (VCC) approach and tool. This chapter summarizes the main contributions of the work presented in this dissertation, identifies future research work and presents some final remarks about the course of the work done.

5.1 Summary of Contributions

The three main contributions of DCC4SPL are:

1. *Product-specific models derivation using VML4RE.* The Variability Modelling Language for Requirements (VML4RE) is both a domain specific language and derivation infrastructure specifically tailored to express how to derive product-specific requirements models. This dissertation shows that VML4RE satisfies the quality attributes of vocabulary familiarity, derivation flexibility, as well as quantifiable attributes such as modularity, stability of the specifications and stability of the configuration knowledge.

2. *Consistency checking between variability model and other models using VCC.* The Variability Consistency Checker (VCC) is a verification approach and tool that supports consistency checking between a variability model (e.g., a feature model), and the models that design the variability units of the variability model (e.g., features). VCC mines constraints between variability units from the variability model and from the

models that design variability. Then, VCC employs propositional formulas to relate all the mined constraints. Checking if all the products in an SPL satisfy consistency constraints is based on searching for a satisfying assignment of a propositional formula. VCC also presents the elements involved in a violation of a consistency rule and explains the cause of the inconsistency. This dissertation shows that VCC satisfies the quality attributes of consistency checking genericity, consistency checking multi-view awareness and consistency checking scalability.

3. *Tool support*. The tool support contributes both research topics: "support to express and perform product-specific models derivation" and "support for consistency checking between variability model and other models". We developed the VML4RE, VCC and "Features Model Metamodel and Editor Tool" prototypes.

5.2 Future Work

From the work we have accomplished in this dissertation, we see several research threads worth investigating.

Self-Adaptative Systems. We plan to extend the applicability of DCC4SPL to the run-time environment of self-adaptative systems. In contrast to development models (such as the ones used in this dissertation), run-time models are used to reason about the operating environment and runtime behaviour, and thus these models must capture abstractions of runtime phenomena. Different dimensions need to be balanced, including resource-efficiency (time, memory, energy), context-dependency (time, location, platform), as well as personalization (quality-of-service specifications, profiles)[1]. The hypothesis is that because models at run-time provide meta-information for these dimensions during execution, run-time decisions can be facilitated and better automated. Thus, according to the reseach community, this technology will play an integral role in the management of self-adaptive systems.

Building Automation Systems. Engineers have made notable progress in creating Building Automation Systems (BASs) to coordinate electrical and mechanical devices to improve comfort and safety of the users of the buildings. Much less is known, however, about verifying consistency rules of BAS specifications (expressed as well-formedness conventions and design patterns) against low-carbon energy-aware constraints and end-users requirements. Existing work in verification does not consider the combinatorial explosion of different requirements imposed by BAS technology, low-carbon energy-aware constraints, and different types of end-users. One of our next steps is to extend DCC4SPL to effectively manage verification of BAS specifications, to improve BAS quality and end-user acceptance. Our solution will provide a low-carbon, energy-aware holistic approach for

[1] http://www.comp.lancs.ac.uk/~bencomo/MRT12/

identifying, modelling, and verifying consistency between multiple specification views, independently of the number and kind of BAS specification languages, underlying implementation technologies, and coping with the combinatorial explosion of requirements.

Tool Support. Our main goal in this dissertation work was to present tool prototypes as proof of concepts. Although our goals did not include the development of commercial tools, we realized that to increase usability of DCC4SPL we need to work more to improve tool support. In particular, we are interested in showing the use of DCC4SPL using other requirements views in a more integrated and usable environment. In this regard, we are considering to develop a graphical concrete syntax for VML4RE and a graphical manager to specify new consistency rules in VCC.

5.3 Final Remarks

There is currently a significant number of requirements modelling, architectural design, software verification and product variants derivation approaches. DCC4SPL undertakes some of the objectives which have been neglected, such as providing a tool-supported approach and language to express derivation of early models using a concrete syntax familiar to developers, and a fully integrated consistency checking approach for the entire SPL specification that harmonizes with the derivation process. There is still room to improve DCC4SPL, in particular in terms of expressing configuration knowledge during derivation specification. The low expressivity of DCC4SPL is due to the lack of designators to point to several different places in the model-based specification using only one language construct. However, any advance in that respect will be paid at the cost of more complexity of the language in terms of number of concepts, relationships and interpreter complexity which will be inevitably more difficult to understand and to be evolved. This reminds us about the dichotomy between general-purpose languages and domain-specific languages, with DCC4SPL we chose a domain-specific languages solution that works very well for its objectives without adding extra (usually) unnecessary complexity to the approach.

The idea for this PhD work was born at in 2009 when the Portuguese government granted me a Ph.D. scholarship. That scholarship was granted in an open competition where the main part of the assessment criteria was the quality of the research proposal. A part of my research proposal was elaborated based on what I learned after writing technical reports and papers, and participating in meetings with the project partners of the European project AMPLE (where I worked as a researcher from 2007 to 2009). Back in 2007 work in verification and model-driven development in early SPLE was then a very novel idea. For years SPLE has been a hot and innovative topic, and many more researchers joined the field in coming years.

Personally, at the end of these years of research, I realize that a work is never strictly finished. There are many things that are left to be done and the Ph.D. project could last

forever. However, my personal goals are achieved and I have acquired new knowledge and gained new experiences. From the challenges that are still left to investigate in the near future, I would definitely choose to invest my energy on contributing to extend DCC4SPL to the topic of models at runtime while improving tool support in terms of usability and stability of the technological solution.

6

Bibliography

[1] Ian F. Alexander and Neil Maiden. *Scenarios, Stories, Use Cases: Through the Systems Development Life-Cycle*. Wiley, 2004.

[2] Germán H. Alférez and Mauricio Alférez. An aspect-oriented framework to model non-functional requirements in software product lines of service-oriented architectures. In Nikola Milanovic, editor, *Non-Functional Properties in Service Oriented Architecture: Requirements, Models and Methods*, chapter 11, pages 246–267. IGI Global, 2011.

[3] Mauricio Alférez, Nuno Amálio, Selim Ciraci, Franck Fleurey, Jörg Kienzle, Jacques Klein, Max E. Kramer, Sébastien Mosser, Gunter Mussbacher, Ella E. Roubtsova, and Gefei Zhang. Aspect-oriented model development at different levels of abstraction. In Robert B. France, Jochen Malte Küster, Behzad Bordbar, and Richard F. Paige, editors, *Modelling Foundations and Applications - 7th European Conference, ECMFA 2011, Birmingham, UK, June 6 - 9, 2011, Proceedings*, volume 6698 of *Lecture Notes in Computer Science*, pages 361–376. Springer, 2011.

[4] Mauricio Alférez, Vasco Amaral, João Araújo, Phil Greenwood, Uirá Kulesza, Ricardo Mateus, Ana Moreira, Afonso Pimentel, Andreas Rummler, Awais Rashid, Rita Ribeiro, and João Santos. Tool suite for aspect-oriented, model-driven requirements engineering. TechReport D1.5, European Project AMPLE, 2009.

[5] Mauricio Alférez, Vasco Amaral, João Araújo, João Santos, Christoph Elsner, Michael C. Jaeger, Uirá Kulesza, Ana Moreira, Afonso Pimentel, Awais Rashid, Rita Ribeiro, João Santos, Christa Schwanninger, and Nathan Weston. Mdd approach

for requirements refinement to architecture. TechReport D1.4, European Project AMPLE, 2009.

[6] Mauricio Alférez, Rodrigo Bonifácio, Leopoldo Teixeira, Paola Accioly, Uirá Kulesza, Paulo Borba, Ana Moreira, and João Araújo. Evaluating approaches for specifying software product line use scenarios. Unpublished Journal Article.

[7] Mauricio Alférez, Uirá Kulesza, Antonielly Garcia, Ana Moreira, João Araújo, and Vasco Amaral. Towards volatility analysis in software product line engineering. In *Second Workshop on Aspect-oriented Product Line Engineering, AOPLE-2, co-located with GPCE 2007, Salzburg, Austria, October 4, 2007, Proceedings*, Lancaster University, Computing Department, TechReport COMP-005-2007, pages 43–46, 2007.

[8] Mauricio Alférez, Uirá Kulesza, Ana Moreira, João Araújo, and Vasco Amaral. Tracing from features to use cases: A model-driven approach. In Patrick Heymans, Kyo Chul Kang, Andreas Metzger, and Klaus Pohl, editors, *Second International Workshop on Variability Modelling of Software-Intensive Systems, Universität Duisburg-Essen, Germany, January 16-18, 2008, Proceedings*, ICB Research Report, pages 81–87, 2008.

[9] Mauricio Alférez, Uirá Kulesza, André Sousa, João Pedro Santos, Ana Moreira, João Araújo, and Vasco Amaral. A model-driven approach for software product lines requirements engineering. In *Proceedings of the Twentieth International Conference on Software Engineering & Knowledge Engineering (SEKE'2008), San Francisco, CA, USA, July 1-3, 2008*, pages 779–784. Knowledge Systems Institute Graduate School, 2008.

[10] Mauricio Alférez, Uira Kulesza, Nathan Weston, João Araújo, Vasco Amaral, Ana Moreira, Awais Rashid, and Michael C. Jaeger. A metamodel for aspectual requirements modelling and composition. TechReport D1.3, European Project AMPLE, 2008.

[11] Mauricio Alférez, Roberto E. Lopez-Herrejon, Ana Moreira, Vasco Amaral, and Alexander Egyed. Ensuring consistency between feature models and model-based specifications - the vcc approach. Unpublished Journal Article.

[12] Mauricio Alférez, Roberto E. Lopez-Herrejon, Ana Moreira, Vasco Amaral, and Alexander Egyed. Supporting consistency checking between features and software product line use scenarios. In Klaus Schmid, editor, *12th International Conference on Software Reuse, ICSR 2011, Pohang, South Korea, June 13-17, 2011, Proceedings*, volume 6727 of *Lecture Notes in Computer Science*, pages 20–35. Springer, 2011.

[13] Mauricio Alférez, Ana Moreira, Vasco Amaral, and João Araújo. Model-driven requirements specification for software product lines. In Janis Osis and Erika Asnina, editors, *Model-Driven Domain Analysis and Software Development: Architectures and Functions*, chapter 17, pages 369–386. IGI Global, 2011.

[14] Mauricio Alférez, Ana Moreira, Uirá Kulesza, João Araújo, Ricardo Mateus, and Vasco Amaral. Detecting feature interactions in spl requirements analysis models. In Sven Apel, William R. Cook, Krzysztof Czarnecki, Christian Kästner, Neil Loughran, and Oscar Nierstrasz, editors, *Proceedings of the First International Workshop on Feature-Oriented Software Development, FOSD 2009, Denver, Colorado, USA, October 6, 2009*, ACM International Conference Proceeding Series, pages 117–123. ACM, 2009.

[15] Mauricio Alférez, João Pedro Santos, Ana Moreira, Alessandro Garcia, Uirá Kulesza, João Araújo, and Vasco Amaral. Multi-view composition language for software product line requirements. In van den Brand et al. [92], pages 103–122.

[16] Sven Apel. *The Role of Features and Aspects in Software Development, Ph.D dissertation*. Ph.d dissertation, Otto-von-Guericke-Universität, Magdeburg, Germany, 2006.

[17] Sven Apel, Wolfgang Scholz, Christian Lengauer, and Christian Kästner. Language-independent reference checking in software product lines. In Sven Apel, Don S. Batory, Krzysztof Czarnecki, Florian Heidenreich, Christian Kästner, and Oscar Nierstrasz, editors, *Proceedings of the Second International Workshop on Feature-Oriented Software Development, FOSD 2010, Eindhoven, Netherlands, October 10, 2010*, pages 65–71. ACM, 2010.

[18] Ivica Aracic, Vaidas Gasiunas, Mira Mezini, and Klaus Ostermann. An overview of caesarj. In Awais Rashid and Mehmet Aksit, editors, *T. Aspect-Oriented Software Development I*, volume 3880 of *Lecture Notes in Computer Science*, pages 135–173. Springer, 2006.

[19] Aspect-Oriented Software Association. Aspect-oriented software development community & conference :: Aosd. http://www.aosd.net.

[20] Bruno Barroca, Levi Lucio, Vasco Amaral, Roberto Félix, and Vasco Sousa. Dsltrans: A turing incomplete transformation language. In Brian A. Malloy, Steffen Staab, and Mark van den Brand, editors, *Software Language Engineering - Third International Conference, SLE 2010, Eindhoven, The Netherlands, October 12-13, 2010, Revised Selected Papers*, volume 6563 of *Lecture Notes in Computer Science*, pages 296–305. Springer, 2010.

[21] Don S. Batory, David Benavides, and Antonio Ruiz Cortés. Automated analysis of feature models: challenges ahead. *Commun. ACM*, 49(12):45–47, 2006.

[22] David Benavides, Sergio Segura, and Antonio Ruiz Cortés. Automated analysis of feature models 20 years later: A literature review. *Inf. Syst.*, 35(6):615–636, 2010.

[23] David Benavides, Sergio Segura, Pablo Trinidad, and Antonio Ruiz Cortés. First international workshop on variability modelling of software-intensive systems, vamos 2007, limerick, ireland, january 16-18, 2007. proceedings. In Klaus Pohl, Patrick

Heymans, Kyo Chul Kang, and Andreas Metzger, editors, *VaMoS*, volume 2007-01 of *Lero Technical Report*, pages 129–134, 2007.

[24] Rodrigo Bonifácio and Paulo Borba. Modeling scenario variability as crosscutting mechanisms. In Sullivan et al. [87], pages 125–136.

[25] Alexandre Bragança and Ricardo Jorge Machado. Automating mappings between use case diagrams and feature models for software product lines. In *Software Product Lines, 11th International Conference, SPLC 2007, Kyoto, Japan, September 10-14, 2007, Proceedings*, pages 3–12. IEEE Computer Society, 2007.

[26] Ruzanna Chitchyan, Phil Greenwood, Américo Sampaio, Awais Rashid, Alessandro F. Garcia, and Lyrene Fernandes da Silva. Semantic vs. syntactic compositions in aspect-oriented requirements engineering: an empirical study. In Sullivan et al. [87], pages 149–160.

[27] Paul Clements and Linda Northrop. *Software Product Lines: Practices and Patterns*. Addison-Wesley, Boston, MA, USA, 2002.

[28] Stephen A. Cook. The complexity of theorem-proving procedures. In Michael A. Harrison, Ranan B. Banerji, and Jeffrey D. Ullman, editors, *Proceedings of the 3rd Annual ACM Symposium on Theory of Computing, May 3-5, 1971, Shaker Heights, Ohio, USA*, pages 151–158. ACM, 1971.

[29] Krzysztof Czarnecki. Overview of generative software development. In Jean-Pierre Banatre, Pascal Fradet, Jean-Louis Giavitto, and Olivier Michel, editors, *Unconventional Programming Paradigms*, volume 3566 of *Lecture Notes in Computer Science*, pages 326–341. Springer Berlin / Heidelberg, 2005.

[30] Krzysztof Czarnecki and Michal Antkiewicz. Mapping features to models: A template approach based on superimposed variants. In *Generative Programming and Component Engineering, 4th International Conference, GPCE 2005, Tallinn, Estonia, September 29 - October 1, 2005, Proceedings*, pages 422–437, 2005.

[31] Krzysztof Czarnecki and Ulrich W. Eisenecker. *Generative programming: methods, tools, and applications*. ACM Press/Addison-Wesley Publishing Co., New York, NY, USA, 2000.

[32] Krzysztof Czarnecki, Simon Helsen, and Ulrich W. Eisenecker. Staged configuration using feature models. In Robert L. Nord, editor, *Software Product Lines, Third International Conference, SPLC 2004, Boston, MA, USA, August 30-September 2, 2004, Proceedings*, volume 3154 of *Lecture Notes in Computer Science*, pages 266–283. Springer, 2004.

[33] Francisco Dantas, Eduardo Figueiredo, Alessandro Garcia, Cláudio Sant'Anna, Uirá Kulesza, Nélio Cacho, Sérgio Soares, Thaís Vasconcelos Batista, Roberta Coelho,

Mauricio Alférez, Ana Moreira, Afonso Pimentel, and João Araújo. Benchmarking stability of aspect-oriented product-line decompositions. In Goetz Botterweck, Stan Jarzabek, Tomoji Kishi, Jaejoon Lee, and Steve Livengood, editors, *Software Product Lines - 14th International Conference, SPLC 2010, Jeju Island, South Korea, September 13-17, 2010. Workshop Proceedings (Volume 2 : Workshops, Industrial Track, Doctoral Symposium, Demonstrations and Tools)*, pages 21–26. Lancaster University, 2010.

[34] Jean-Marc DeBaud and Klaus Schmid. A systematic approach to derive the scope of software product lines. In *ICSE*, pages 34–43, 1999.

[35] Edsger W. Dijkstra. On the role of scientific thought. Unpublished work, August 1974.

[36] Edsger W. Dijkstra. On the role of scientific thought. In *Selected Writings on Computing: A Personal Perspective*, pages 60–66. Springer-Verlag, 1982.

[37] Marc Eaddy, Alfred Aho, and Gail C. Murphy. Identifying, assigning, and quantifying crosscutting concerns. In *ACoM '07: Proceedings of the First International Workshop on Assessment of Contemporary Modularization Techniques*, page 2, Washington, DC, USA, 2007. IEEE Computer Society.

[38] Marc Eaddy, Thomas Zimmermann, Kaitlin D. Sherwood, Vibhav Garg, Gail C. Murphy, Nachiappan Nagappan, and Alfred V. Aho. Do crosscutting concerns cause defects? *IEEE Trans. Softw. Eng.*, 34:497–515, July 2008.

[39] Steve Easterbrook, Anthony Finkelstein, Jeff Kramer, and Bashar Nuseibeh. Coordinating distributed viewpoints: the anatomy of a consistency check. *Concurrent Engineering: Research and Applications*, 2:209–222, 1994.

[40] Alexander Egyed. Fixing inconsistencies in uml design models. In *29th International Conference on Software Engineering (ICSE 2007), Minneapolis, MN, USA, May 20-26, 2007*, pages 292–301. IEEE Computer Society, 2007.

[41] Magnus Eriksson, Jürgen Börstler, and Kjell Borg. The pluss approach - domain modeling with features, use cases and use case realizations. In J. Henk Obbink and Klaus Pohl, editors, *Software Product Lines, 9th International Conference, SPLC 2005, Rennes, France, September 26-29, 2005, Proceedings*, volume 3714 of *Lecture Notes in Computer Science*, pages 33–44. Springer, 2005.

[42] Magnus Eriksson, Jürgen Börstler, and Kjell Borg. Managing requirements specifications for product lines - an approach and industry case study. *Journal of Systems and Software*, 82(3):435 – 447, 2009.

[43] Eduardo Figueiredo, Nélio Cacho, Cláudio Sant'Anna, Mario Monteiro, Uirá Kulesza, Alessandro Garcia, Sérgio Soares, Fabiano Cutigi Ferrari, Safoora Shakil Khan, Fernando Castor Filho, and Francisco Dantas. Evolving software product lines with

aspects: an empirical study on design stability. In Wilhelm Schäfer, Matthew B. Dwyer, and Volker Gruhn, editors, *30th International Conference on Software Engineering (ICSE 2008*, pages 261–270. ACM, 2008.

[44] Robert E. Filman, Tzilla Elrad, Siobhán Clarke, and Mehmet Akşit, editors. *Aspect-Oriented Software Development*. Addison-Wesley, Boston, 2005.

[45] Hassan Gomaa. *Designing Software Product Lines with UML: From Use Cases to Pattern-Based Software Architectures*. Addison Wesley Longman Publishing Co., Inc., Redwood City, CA, USA, 2004.

[46] Jack Greenfield, Keith Short, Steve Cook, and Stuart Kent. *Software Factories: Assembling Applications with Patterns, Models, Frameworks, and Tools*. John Wiley & Sons, 2004.

[47] David Harel and Bernhard Rumpe. Meaningful modeling: What's the semantics of "semantics"? *IEEE Computer*, 37(10):64–72, 2004.

[48] Florian Heidenreich, Jan Kopcsek, and Christian Wende. Featuremapper: mapping features to models. In *Companion of the 30th international conference on Software engineering*, ICSE Companion '08, pages 943–944, New York, NY, USA, 2008. ACM.

[49] Florian Heidenreich, Pablo Sánchez, João Santos, Steffen Zschaler, Mauricio Alférez, João Araújo, Lidia Fuentes, Uirá Kulesza, Ana Moreira, and Awais Rashid. Relating feature models to other models of a software product line - a comparative study of featuremapper and vml*. *T. Aspect-Oriented Software Development*, 7:69–114, 2010.

[50] Ivar Jacobson. *Object-Oriented Software Engineering: A Use Case Driven Approach*. Addison Wesley Longman Publishing Co., Inc., Redwood City, CA, USA, 2004.

[51] Praveen K. Jayaraman, Jon Whittle, Ahmed M. Elkhodary, and Hassan Gomaa. Model composition in product lines and feature interaction detection using critical pair analysis. In Gregor Engels, Bill Opdyke, Douglas C. Schmidt, and Frank Weil, editors, *Model Driven Engineering Languages and Systems, 10th International Conference, MoDELS 2007, Nashville, USA, September 30 - October 5, 2007, Proceedings*, volume 4735 of *Lecture Notes in Computer Science*, pages 151–165. Springer, 2007.

[52] Frédéric Jouault and Ivan Kurtev. Transforming models with atl. In Jean-Michel Bruel, editor, *Satellite Events at the MoDELS 2005 Conference, MoDELS 2005 International Workshops, Doctoral Symposium, Educators Symposium, Montego Bay, Jamaica, October 2-7, 2005, Revised Selected Papers*, volume 3844 of *Lecture Notes in Computer Science*, pages 128–138. Springer, 2005.

[53] Kyo C. Kang, Sholom G. Cohen, James A. Hess, William E. Novak, and A. Spencer Peterson. Feature-oriented domain analysis (foda) feasibility study. TechReport

CMU/SEI-90-TR-021, Carnegie-Mellon University Software Engineering Institute, 1990.

[54] Shmuel Katz, Mira Mezini, and Jörg Kienzle, editors. *Transactions on Aspect-Oriented Software Development VII - A Common Case Study for Aspect-Oriented Modeling*, volume 6210 of *Lecture Notes in Computer Science*. Springer, 2010.

[55] Joerg Kienzle, Nicolas Guelfi, and Sadaf Mustafiz. Crisis management systems: A case study for aspect-oriented modeling, June 2009. `http://www.cs.mcgill.ca/ joerg/taosd/TAOSD/TAOSD.html`.

[56] Jörg Kienzle, Nicolas Guelfi, and Sadaf Mustafiz. Crisis management systems: A case study for aspect-oriented modeling. In *T. Aspect-Oriented Software Development* [54], pages 1–22.

[57] Tomoji Kishi and Natsuko Noda. Formal verification and software product lines. *Commun. ACM*, 49(12):73–77, 2006.

[58] Anneke Kleppe. *Software Language Engineering: Creating Domain-Specific Languages Using Metamodels*. Addison-Wesley Professional, 1 edition, 2008.

[59] Anneke G. Kleppe, Jos Warmer, and Wim Bast. *MDA Explained: The Model Driven Architecture: Practice and Promise*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2003.

[60] Jasna Kovacevic, Mauricio Alférez, Uirá Kulesza, Ana Moreira, João Araújo, Vasco Amaral, Vander Alves, Awais Rashid, and Ruzanna Chitchyan. Survey of the state-of-the-art in requirements engineering for software product line and model-driven requirements engineering. TechReport D1.1, European Project AMPLE, 2007.

[61] Philippe Kruchten. The 4+1 view model of architecture. *IEEE Software*, 12(6):42–50, 1995.

[62] Philippe Kruchten. *The Rational Unified Process: An Introduction*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 3 edition, 2003.

[63] Charles W. Krueger. Software reuse. *ACM Comput. Surv.*, 24(2):131–183, 1992.

[64] Uirá Kulesza, Cláudio Sant'Anna, Alessandro Garcia, Roberta Coelho, Arndt von Staa, and Carlos José Pereira de Lucena. Quantifying the effects of aspect-oriented programming: A maintenance study. In *22nd IEEE International Conference on Software Maintenance (ICSM 2006), 24-27 September 2006, Philadelphia, Pennsylvania, USA*, pages 223–233. IEEE Computer Society, 2006.

[65] Martin Lippert and Cristina Videira Lopes. A study on exception detecton and handling using aspect-oriented programming. In Carlo Ghezzi, Mehdi Jazayeri, and Alexander L. Wolf, editors, *Proceedings of the 22nd International Conference on on*

Software Engineering, ICSE 2000, Limerick Ireland, June 4-11, 2000, pages 418–427. ACM, 2000.

[66] Roberto Erick Lopez-Herrejon and Alexander Egyed. Detecting inconsistencies in multi-view models with variability. In Thomas Kühne, Bran Selic, Marie-Pierre Gervais, and François Terrier, editors, *Modelling Foundations and Applications, 6th European Conference, ECMFA 2010, Paris, France, June 15-18, 2010. Proceedings*, volume 6138 of *Lecture Notes in Computer Science*, pages 217–232. Springer, 2010.

[67] Francisco J. Lucas, Fernando Molina, and José Ambrosio Toval Álvarez. A systematic review of uml model consistency management. *Information & Software Technology*, 51(12):1631–1645, 2009.

[68] Levi Lucio, Bruno Barroca, and Vasco Amaral. A technique for automatic validation of model transformations. In Dorina C. Petriu, Nicolas Rouquette, and Øystein Haugen, editors, *Model Driven Engineering Languages and Systems - 13th International Conference, MODELS 2010, Oslo, Norway, October 3-8, 2010, Proceedings, Part I*, volume 6394 of *Lecture Notes in Computer Science*, pages 136–150. Springer, 2010.

[69] Slavisa Markovic and Thomas Baar. Refactoring ocl annotated uml class diagrams. *Software and System Modeling*, 7(1):25–47, 2008.

[70] Joaquin Miller and Jishnu Mukerji. Mda guide version 1.0.1. TechReport omg/2003-06-01, Object Management Group, 2003.

[71] Hugo Morganho, Catarina Gomes, João P. Pimentão, Rita Ribeiro, Birgit Grammel, Christoph Pohl, Andreas Rummler, Christa Schwanninger, Ludger Fiege, and Michael Jaeger. Requirement specifications for industrial case studies. TechReport D5.2, European Project AMPLE, 2008.

[72] Andrew Moss and Henk L. Muller. Efficient code generation for a domain specific language. In Robert Glück and Michael R. Lowry, editors, *Generative Programming and Component Engineering, 4th International Conference, GPCE 2005, Tallinn, Estonia, September 29 - October 1, 2005, Proceedings*, volume 3676 of *Lecture Notes in Computer Science*, pages 47–62. Springer, 2005.

[73] Bashar Nuseibeh, Jeff Kramer, and Anthony Finkelstein. A framework for expressing the relationships between multiple views in requirements specification. *IEEE Trans. Software Eng.*, 20(10):760–773, 1994.

[74] Jon Oldevik, Øystein Haugen, and Birger Møller-Pedersen. Confluence in domain-independent product line transformations. In Marsha Chechik and Martin Wirsing, editors, *Fundamental Approaches to Software Engineering, 12th International Conference, FASE 2009, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2009, York, UK, March 22-29, 2009. Proceedings*, volume 5503 of *Lecture Notes in Computer Science*, pages 34–48. Springer, 2009.

[75] OMG. Meta object facility (mof) 2.0 query/view/transformation specification. TechReport formal/2011-01-01, Object Management Group, 2011.

[76] D. L. Parnas. On the design and development of program families. *IEEE Trans. Softw. Eng.*, 2:1–9, January 1976.

[77] David Lorge Parnas. Designing software for ease of extension and contraction. *IEEE Trans. Software Eng.*, 5(2):128–138, 1979.

[78] Afonso Pimentel, Rita Ribeiro, Ana Moreira, João Araújo, João Santos, António Costa, Mauricio Alférez, and Uirá Kulesza. Hybrid assessment method for software product lines. In Awais Rashid, Jean-Claude Royer, and Andreas Rummler, editors, *Aspect-Oriented, Model-Driven Software Product Lines: The AMPLE Way*, chapter 5, pages 125–158. Cambridge University Press, 2011.

[79] Klaus Pohl, Günter Böckle, and Frank J. van der Linden. *Software Product Line Engineering: Foundations, Principles and Techniques*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2005.

[80] Jeff Rothenberg. The nature of modeling. In Lawrence E. Widman, Kenneth A. Loparo, and Norman R. Nielsen, editors, *AI, Simulation & Modeling*, AI, Simulation & Modeling, pages 75–92. John Wiley & Sons, Inc., New York, NY, USA, August, 1989. Reprinted as N-3027-DARPA, The RAND Corporation, November 1989.

[81] Grzegorz Rozenberg, editor. *Handbook of Graph Grammars and Computing by Graph Transformations, Volume 1: Foundations*. World Scientific, 1997.

[82] Knowledge Systems Institute Graduate School, editor. *Proceedings of the Twentieth International Conference on Software Engineering & Knowledge Engineering (SEKE'2008), San Francisco, CA, USA, July 1-3, 2008*. KSI, 2008.

[83] Shane Sendall and Wojtek Kozaczynski. Model transformation: The heart and soul of model-driven software development. *IEEE Software*, 20(5):42–45, 2003.

[84] Ida Solheim and Ketil Solen. Technology research explained. TechReport SINTEF A313, SINTEF ICT, March 2007.

[85] David Steinberg, Frank Budinsky, Marcelo Paternostro, and Ed Merks. *EMF: Eclipse Modeling Framework 2.0*. Addison-Wesley Professional, 2nd edition, 2009.

[86] Vijayan Sugumaran, Sooyong Park, and Kyo C. Kang. Introduction: Software product line engineering. *Commun. ACM*, 49:28–32, December 2006.

[87] Kevin J. Sullivan, Ana Moreira, Christa Schwanninger, and Jeff Gray, editors. *Proceedings of the 8th International Conference on Aspect-Oriented Software Development, AOSD 2009, Charlottesville, Virginia, USA, March 2-6, 2009*. ACM, 2009.

[88] Marta-Silvia Tabares, Germán-Harvey Alférez, and Mauricio Alférez. Aspect-oriented software development: A practical case for an on-line help desk system. *Revista Avances en Sistemas e Informática*, 5(2):61–68, 2008.

[89] Gabriele Taentzer. Agg: A graph transformation environment for modeling and validation of software. In John L. Pfaltz, Manfred Nagl, and Boris Böhlen, editors, *Applications of Graph Transformations with Industrial Relevance, Second International Workshop, AGTIVE 2003, Charlottesville, VA, USA, September 27 - October 1, 2003, Revised Selected and Invited Papers*, volume 3062 of *Lecture Notes in Computer Science*, pages 446–453. Springer, 2003.

[90] Sahil Thaker, Don S. Batory, David Kitchin, and William R. Cook. Safe composition of product lines. In Charles Consel and Julia L. Lawall, editors, *Generative Programming and Component Engineering, 6th International Conference, GPCE 2007, Salzburg, Austria, October 1-3, 2007, Proceedings*, pages 95–104. ACM, 2007.

[91] Muhammad Usman, Aamer Nadeem, Tai-hoon Kim, and Eun-suk Cho. A survey of consistency checking techniques for uml models. In *Proceedings of the 2008 Advanced Software Engineering and Its Applications*, ASEA '08, pages 57–62, Washington, DC, USA, 2008. IEEE Computer Society.

[92] Mark van den Brand, Dragan Gasevic, and Jeff Gray, editors. *Software Language Engineering, Second International Conference, SLE 2009, Denver, CO, USA, October 5-6, 2009, Revised Selected Papers*, volume 5969 of *Lecture Notes in Computer Science*. Springer, 2010.

[93] Jilles van Gurp, Jan Bosch, and Mikael Svahnberg. On the notion of variability in software product lines. In *2001 Working IEEE / IFIP Conference on Software Architecture (WICSA 2001), 28-31 August 2001, Amsterdam, The Netherlands*, pages 45–54. IEEE Computer Society, 2001.

[94] Markus Volter and Thomas Stahl. *Model-Driven Software Development*. Wiley, Glasgow, UK, 2006.

[95] Trevor J. Young and Trevor J. Young. Using aspectj to build a software product line for mobile devices. Masterthesis, University of British Columbia, British Columbia, Canada, 2005.

[96] Steffen Zschaler, Pablo Sanchez, João Santos, Mauricio Alférez, Ana Moreira, João Araújo, Uira Kulesza, and Lidia Fuentes. Variability management. In Awais Rashid, Jean-Claude Royer, and Andreas Rummler, editors, *Aspect-Oriented, Model-Driven Software Product Lines: The AMPLE Way*, chapter 4, pages 82–124. Cambridge University press, 2011.

[97] Steffen Zschaler, Pablo Sánchez, João Pedro Santos, Mauricio Alférez, Awais Rashid, Lidia Fuentes, Ana Moreira, João Araújo, and Uirá Kulesza. Vml* - a family of

languages for variability management in software product lines. In van den Brand et al. [92], pages 82–102.

Part II

Research Papers

7

Introduction

This chapter presents and includes a copy of some of the book chapters and peer-reviewed papers written for international events.

- Chapter 8 - A Model-Driven Approach for Software Product Lines Requirements Engineering [9] (Page 119).
- Chapter 9 - Multi-View Composition Language for Software Product Line Requirements [15] (Page 127).
- Chapter 10 - VML* – A Family of Languages for Variability Management in Software Product Lines [97] (Page 149).
- Chapter 11 - Model-Driven Requirements Specification for Software Product Lines [13] (Page 173).
- Chapter 12 - Evaluating Approaches for Specifying Software Product Line Use Scenarios [6] (Page 187).
- Chapter 13 - Supporting Consistency Checking between Features and Software Product Line Use Scenarios [12] (Page 215).
- Chapter 14 - Ensuring Consistency Between Feature Models and Model-Based Specications - The VCC Approach [11] (Page 233).

Also, there are other publications that helped to shape the work presented in this dissertation. However, we chose to show here those with larger impact on the research contributions of this PhD dissertation. Showing all the publications here make this work

too extensive. The following are the published papers that were not selected to appear here:

- Journal paper
  - Relating Feature Models to Other Models of a Software Product Line - A Comparative Study of FeatureMapper and VML* [49].
  - Aspect-Oriented Software Development: A Practical Case for an On-line Help Desk System [88].
- Book chapters
  - Variability Management [96].
  - Hybrid Assessment Method for SPL [78].
  - An Aspect-Oriented Framework to Model Non-Functional Requirements in Software Product Lines of Service-Oriented Architectures [2].
- International conference paper
  - Aspect-Oriented Model Development at Different Levels of Abstraction [3].
- Workshop papers
  - Tracing between Features and Use Cases: A Model-Driven Approach [8].
  - Detecting Feature Interactions in SPL Requirements Analysis Models [14].
  - Benchmarking Stability of Aspect-Oriented Product-Line Decompositions [33].
- Technical reports:
  - Survey of State-of-the-Art in Requirements Engineering for Software Product Lines and Model-Driven Requirements Engineering [60].
  - A Metamodel for Aspectual Requirements Modelling and Composition [10].
  - Tool Suite for Aspect-Oriented, Model-Driven Requirements Engineering [4].
  - MDD Approach for Requirements Refinement to Architecture [5].

8

A Model-Driven Approach for Software Product Lines Requirements Engineering

Authors: Mauricio Alférez, Uirá Kulesza, André Sousa, João Santos, Ana Moreira, João Araújo, Vasco Amaral.

Paper Summary: This paper presents a model-driven approach for variability management in product lines that addresses traceability between features and UML requirements models. As a proof of concept we used UML use case and activity diagrams and features models to specify SPL requirements. The main contribution was to show how model-driven techniques can be used as a base to automatically derive requirements models for specific products of a SPL, and views that explicitly illustrate the relationships between features and UML requirements model elements. The model-driven techniques applied in our approach were to apply bindings between metamodels, create a tracing metamodel strategy based on proxy metaclasses that allow a n-to-m (were n, m >=0) mapping between features and model elements in use case and activity diagrams, generate specific product requirements models automatically, and use composition rules to specify compositions between use cases by means of their respective activity diagrams. Another main contribution of this paper was to describe the process from requirements specified textually in sentences to features and use cases, and the way use cases and their respective activity diagrams should be composed using a composition language. However, we delay the explanaition of the composition language to Chapter 9 - Multi-View Composition Language for Software Product Line Requirements.

Authors Contribution: Mauricio Alférez was the main author and responsible for the main part of the research and writing of this paper, accounting for the 90% of the work. Other authors gave interesting comments that helped to improve the content of the paper.

Publication: Published in the proceedings of the Twentieth International Conference on Software Engineering & Knowledge Engineering (SEKE'2008), San Francisco, CA, USA, July 1-3, 2008 [9, 82]. Acceptance rate (full papers): 36%. Conference classification: CORE B.

A Model-Driven Approach for Software Product Lines Requirements Engineering

Mauricio Alférez, Uirá Kulesza, André Sousa, João Santos,
Ana Moreira, João Araújo, Vasco Amaral
Dept. Informática, FCT, Universidade Nova de Lisboa, Portugal
{mauricio.alferez, uira, als, jps, amm, ja, vasco.amaral}@di.fct.unl.pt

Abstract

UML and feature models complement each other well and can be the base techniques for a systematic method to identify and model software product line (SPL) requirements. In this paper, we present a model-driven approach to trace both features and UML requirements analysis model elements, and to automatically derive valuable models for domain and application engineering. The resulting contribution is a synergetic approach for SPL requirements. We illustrate it by using a home automation system product line.

1. Introduction

Software product line (SPL) approaches [1-3] aim at improving the productivity and quality of software development by enabling the management of common and variable features of a system family. A system family is defined as a set of programs that shares common functionalities and maintains specific functionalities that vary according to specific family product members. A SPL can be seen as a system family that addresses a specific market segment [1].

Over the past few years, several SPL development approaches have been proposed [1-4]. Most of them motivate the identification of common and variable features of the SPL by means of domain analysis activities. A feature [4] can be seen as a system property or functionality that is relevant to some stakeholder and is used to capture commonalities or discriminate among products in a SPL. SPL features are typically represented in domain analysis using feature models [5]. Other requirements models (e.g., use case and activity models) can be used to better describe and detail the SPL requirements. The feature and requirements models are then used as a reference along all the process to guide the development of the SPL.

Some research works have addressed the use of feature models in combination with other models. Approaches like [6] and [7] propose to create relationships between features and UML models by means of intrusive graphical elements such as, presence conditions or notes to indicate variability. The main disadvantage of these approaches is the creation of convoluted and polluted models, which bring difficulties to understand, maintain and scale the models and trace links between features and UML elements.

Other approaches [3, 8, 9] give some directions on how to model and trace variability information. However, and similarly to what happens with the previous approaches, they do not provide specific activities and tool support for modeling, tracing and generating requirements models for specific products based on the tracing information.

This paper presents a model-driven approach for variability management in product lines that addresses traceability between features and UML requirements models (like use cases and activity models). The main contribution is to show how model-driven techniques can be used to automatically derive, from the information provided by the trace links, requirements models for specific products of a SPL, and views that explicitly illustrate the relationships between features and UML requirements model elements. These views are useful in both domain and application engineering stages. The general idea of our approach is to apply bindings between metamodels, create a simple tracing metamodel strategy, generate specific product requirements models automatically, and use composition rules to specify compositions between use cases by means of their respective activity diagrams.

This paper starts with an overview of our metamodelling strategy and approach main activities in

Originally published in the proceedings of the Twentieth International Conference on Software Engineering & Knowledge Engineering (SEKE'2008), San Francisco, CA, USA, July 1-3, 2008, pages 779–784. Knowledge Systems Institute Graduate School, 2008.

Section 2. These activities are illustrated using a home automation system, in Section 3. Section 4 explores and presents lessons learned from the application of our approach. Finally, Section 5 concludes the paper and points out directions to some future work.

2. A Model-Driven Approach for SPL Requirements Engineering

Traceability between feature and requirements models is supported in our approach by a metamodelling strategy. Figure 1(a) introduces the adopted metamodelling strategy and Figure 1(b) makes that strategy concrete through feature, use case and activity metamodels.

A variability model is used to represent the common and variable SPL features. One or more requirements models detail the SPL requirements. A traceability metamodel is used to link abstractions from the variability and the requirements models. This enables the navigation across abstractions of the different types of models using model-driven techniques and tools. The traceability model also supports backward and forward traceability between a feature model (or any of its configurations) and requirements models. Each configuration defines the features of a specific product from the SPL.

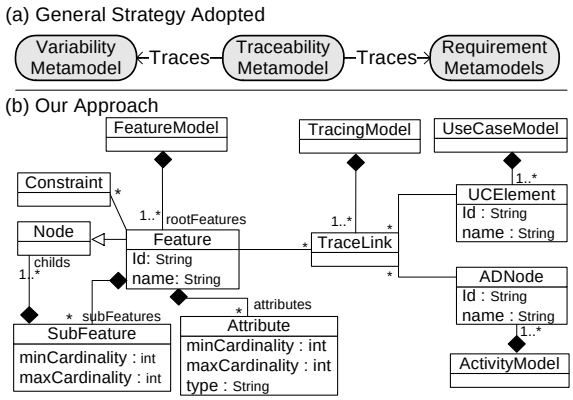


Figure 1. Traceability support strategy

Our approach adopts a feature metamodel based on [7] as the variability model. UML use case and activity models specify the SPL requirements. Due to the large dimension of the their metamodels, we only show the use case model element "UCElement" and activity diagram node "ADNode" from which all the traceable elements of each model can be inherited. Activity diagrams model the behavior of use cases. Use cases and activity models are related to each other by means of the feature to which they are connected.

Our metamodel (Figure 1(b)) supports the set of models that we create in domain and application engineering. The metamodel enables the creation of models in *conformance to* their respective metamodels [10], and helps to understand the relationships between the models elements. Besides the metamodelling strategy, our approach also defines a set of systematic activities in the domain and application engineering stages. The SPL requirements models are created and manipulated during these stages using model-driven techniques and tools.

At the domain analysis level, we perform the activities described next. Although they are organized sequentially, they are typically executed iteratively and incrementally.

1. Identify requirements. The SPL requirements can be elicited using traditional requirements engineering techniques such as inspection of existing documents that describe the problem domain, existing catalogues [11], stakeholders interview transcripts or by using mining techniques [12]. Other approaches such as [9] and [3] already address this activity in detail.

2. Group requirements into features. During this activity, we organize the SPL requirements into clusters according to the specific SPL features they are related to. There are semi-automatic clustering techniques such as [13] that could help to support this activity. However, the specific steps followed in the clustering sub-process are out of the scope of this paper and are not included due to lack of space.

3. Refactor requirements and features. During the previous activity, requirements could result to be linked to more than one feature. We propose to refactor those requirements to be ideally related to only one feature, whenever possible. It contributes to achieve a better modularization of the SPL requirements through the separation of the variable parts of each requirement [14] as well as facilitate establishing tracing links between requirements and features.

4. Model SPL features and use cases. This activity structures and represents the SPL requirements using use case and feature models. Use case models specify the functional requirements and feature models specify the SPL features and variability-commonality information.

5. Relate features to use cases. The relationships between features and use cases are specified visually in a table of trace links. The table allows defining and maintaining the trace relationships between features and UML elements.

6. Generate SPL use cases annotated with features. A model-driven tool developed for our approach uses the relationships between use cases and features to generate specific use case models annotated with features [15]. In the annotated model, each use case is shown with the respective(s) feature(s) related to it. Therefore, it is also possible to obtain the set of use cases related to a specific feature. This allows the domain analysis engineers and SPL architects to reason about how each use case is related to the SPL features and to analyze the impact of change of specific features in SPL requirements.

7. Model use cases as activity diagrams. The detailed behavior of each use case is modeled using activity diagrams, similarly to what happens in several UML-based methods, such as RUP [16]. Use cases specified as activity diagrams, in contrast with textual-based specifications, allows us to enable the use of model-driven generation tools by providing models that conform to a metamodel (i.e., UML activity diagram metamodel) and to help to avoid ambiguity in the specifications [3]. The detailed specification of use cases as activity models also enables us to customize the behavior of use cases according to the features selected to a specific SPL configuration.

8. Specify composition rules between use cases. Each composition rule defines how a variable use case (i.e., linked to a variable feature) can interfere or modify the normal execution of a mandatory use case (i.e., linked to a common feature). Composition rules are defined in terms of the elements of the activity diagrams (e.g., activities, initial state or final state).

The models produced during domain engineering are used in application engineering to generate use case and activity models for specific SPL configurations. We define three activities in application engineering:

1. Define a SPL configuration. The application engineer specifies a SPL configuration, where s/he chooses which optional and alternative features are going to be part of the final application.

2. Generate a use case model from a SPL configuration. Our tool [15] generates the use case model related to the SPL configuration defined in the previous activity. The input for the generation is the SPL use case model, the SPL configuration and the table that maintains the trace links between features and use cases.

3. Generate activity diagrams from a SPL configuration. Our tool is also used to generate activity diagrams related to a specific SPL configuration. In this process, the original activity diagrams can be composed using the composition rules defined in the domain engineering stage. The choice of which composition rules will be used is based on the features included in the SPL configuration. The activity diagram of each extension use case, for example, can be composed with mandatory use cases if the variable feature related to it was selected by the application engineer (step 1 of application engineering).

3. Applying the Approach to a Case Study

To illustrate the activities described in the previous section, we have chosen a home automation system, called Smart Home (see also [3]). This system is one of the SPL case studies proposed by the industrial partners of the European project AMPLE [17]; due to its complexity, we will focus only on a subset of the Security module.

The requirements and feature identification, and refactoring activities, are described in [18]. They provided the features and requirements of our case study. By inspecting those requirements and features, we modeled the SPL feature and use case models. Figure 2 shows the most relevant artifacts produced by the activities of our approach. It shows how each artifact produced in the domain engineering perspective is used to create or derive other artifacts for a specific product in the application engineering perspective. Next we describe the domain engineering activities from our approach.

Model SPL features and use cases. Figure 2(a) shows the feature model of our Security module. It has three main features: *Room Surveillance*, *Admittance Control* and *Intrusion Detection*. *Room Surveillance* is an optional feature that includes *Indoor Camera Surveillance* and, optionally, *Indoor Motion Detection*. The inhabitant can be admitted to enter the house after passing either a *Biometrical Analysis*, *Smart Card*, or entering a *PIN*. In case of selecting intrusion detection, the *Glass Break Detection* must be included and optionally, motion detection sensors and cameras for outdoor security. The notation used in Figure 2(a) is described in Figure 2(j).

We can obtain the SPL use cases from the requirements and features previously identified. Use case modeling is used to better structure the SPL requirements and add more semantics to the features [6]. Figure 2(b) shows the use case model of the case study. The initial SPL features and use cases can be refined and incremented to consider new variabilities or products that need to be included in the family. Both use case and feature models must be updated when

new features are considered or existing ones need to be modified or removed.

Relate features to use cases and generate SPL use case models annotated with features. So that traceability can be maintained between use cases and features, we define an activity to specify the trace relationships between those artifacts. By inspecting the requirements and features of the case study, we related, for example, the *Open Front Door* use case with *Admittance Control*, refined into *Biometrical Analysis*, *Smart Card*, and *PIN* features because to open the front door, the system requires *Admittance Control.* Figure 2(c) shows one of the views that can be generated using the trace links relationships between use cases and features. The open branch in the tree-like structure shows that the *Smart Card* feature is related with the use cases *Identify User by Smart Card*, *Open Front Door* and *Configure Security Management*.

The traceability views of the relationships between features and other artifacts allow the domain analysis engineers and SPL architects to reason about the domain analysis artifacts interdependencies. Currently, there are two kinds of traceability views that our approach can generate in this activity: (i) A use case model annotated with the respective related features; and (ii) a tree structure that shows the list of use cases with the related features and, optionally, the list of features with the related use cases (as in Figure 2(c)).

Create activity diagrams. The behavior of each use case can be specified in our approach using activity diagrams. These diagrams were created by inspecting the requirements. Figure 2(e) and Figure 2(f) shows, for example, the activity diagrams of the *Identify User by Smart Card* and *Identify User* use cases.

Specify composition rules between use cases. Use cases composition is addressed in our approach by means of a set of composition rules. Each composition rule defines how a use case can interfere, modify, or replace the execution of another use case. The composition rules are defined in terms of activity diagrams elements (i.e., activities, initial and final nodes). Composition rules are used during the application engineering phase to derive the specific behavior of use cases for a SPL configuration or product. Figure 2(d) presents the composition rule between the use cases *Identify User* and *Identify User by Smart Card*. It shows how the *Identify User by SmartCard* use case can modify the *Identify User* use case to include additional steps related to the *Smart Card* variable feature. The application of the composition rule is shown in the following subsection where specific activity diagrams can be generated for each product of the SPL.

Next, we describe the execution of the application engineering activities of our approach in the context of the Smart Home case study.

Define a SPL configuration and generate the related use cases and activity diagrams. The first activity in application engineering is to specify a SPL configuration to decide which features will be part of the final application. Figure 2(g) shows a configuration of the case study feature model shown in Figure 2(a) (see the notation used in Figure 2(j)).

Based on the feature model configuration, the relationships between use cases and features, and the SPL use case model (Figure 2(b)), a use case model can be automatically derived using the tool from our approach [15]. Figure 2(h) shows the use case model of the product specified in Figure 2(g).

The final activity of application engineering in our approach involves the automatic customization of the activity diagrams related to each of the SPL use cases using the composition rules specified in domain engineering. Only the activity diagrams of the use cases that are part of the SPL configuration are customized. Figure 2(i) shows the composition between the activity diagrams that describe the *Identify User* and *Identify User by Smart Card* use cases depicted in Figure 2(f) and Figure 2(e), using a `Replace with` composition rule depicted in Figure 2(d). It is not the aim of this paper to present a full-fledged composition language; we just show how it would look like. A complete composition language is one of our aims for future work. For additional details about the current version of our composition language, please refer to [17].

4. Benefits and Lessons Learned

In the context of the European project AMPLE, experiments with our approach have shown that the information of the relationships among the SPL requirements models can be used to support: (i) forward and backward traceability between features and requirements models like use case and activity models; and (ii) reasoning about the impact of feature interactions in the SPL requirements (expressed by the use cases and activity diagrams). Forward and backward traceability enables the creation of tracing queries over all the requirements artifacts and the derivation of specific requirements models for a determined product in the SPL using an model-driven derivation tool, as the one that we have developed [15]. In addition, it enables to the developers to visualize the features changes effects in the SPL requirements through the automatic modification of the

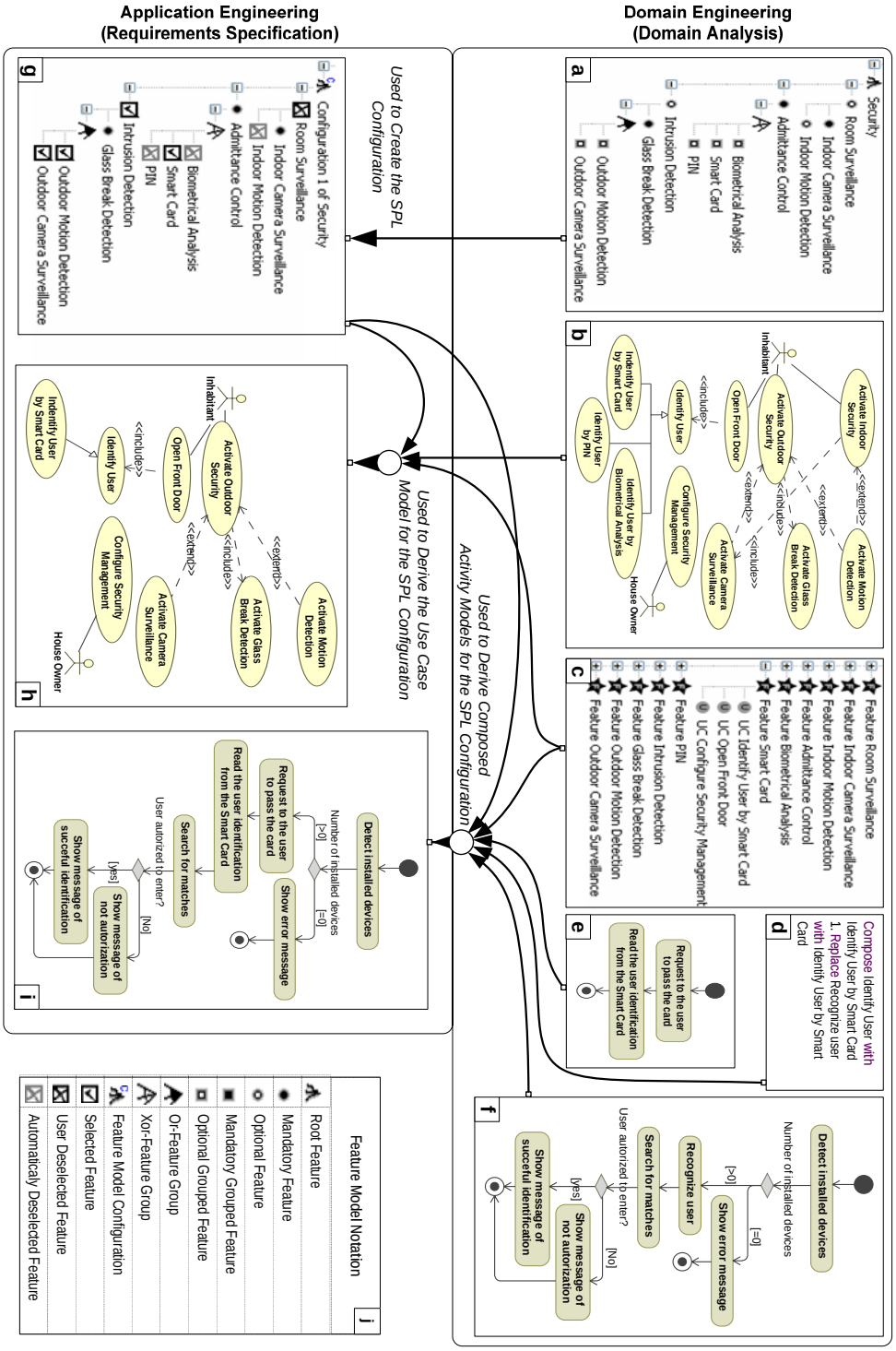


Figure 2. Some of the artifacts produced in the Smart Home Security module case study. (a) SPL feature model; (b) SPL use case model; (c) Use cases related to features; (d) Composition rule between "Identify User" and "Identify User by Smart Card"; (e) Activity diagram of the "Identify User by Smart Card" use case; (f) Activity diagram of the "Identify User" use case; (g) Configuration of the SPL feature model; (h) Use case model for a specific product; (i) Composing "Identify User" with "Identify User by Smart Card"; (j) Feature model notation.

models. On the other hand, the information about feature interactions offered by our approach is useful during the design of SPL architectures to allow an adequate modularization and implementation of their respective features. However, in this paper we have only concentrated on describing the traceability functionalities.

Our metamodelling strategy (Section 2) brings the following benefits to the definition of our approach: (i) *simplicity* – the integration between the metamodels of the feature and requirements models is very easy to understand and evolve; and (ii) *flexibility* – the strategy can be applied to any requirements notation that has a well-defined metamodel.

Our approach also enables composition of crosscutting use cases by representing their steps in activity diagrams. Composition rules are used to specify how the behavior of a use case can affect the behavior of another one. We believe this is an effective way to represent how the SPL variabilities occur along the use cases behavior. The integrated use of these activity diagrams, composition rules and a SPL configuration allows generating the specific behavior of a SPL product. The resulting activity diagram representing the use cases of a product can then be used with different purposes, such as, for example, to document the final requirements of the product or to generate specific test cases for the product.

5. Conclusions and Future Work

This paper presented a model-driven approach to model, specify and trace SPL features and requirements supported by an automated tool. We adopted a simple but useful metamodel integration strategy to allow tracing between features and other requirements models. The approach includes domain and application engineering activities, both illustrated using the Smart Home SPL case study.

Our work is currently being extended to address additional perspectives, such as: (i) to provide more explicit guidance for non-functional requirements and feature interactions modeling and to create special trace views for these concerns; (ii) to deal with uncertainty or volatile requirements in SPLs; (iii) to continue exploiting the activity diagrams to model scenarios [19]; and (iv) define a more complete approach in the context of the AMPLE project to provide tracing support from features and requirements models to artifacts of later software development stages, such as, architecture models and source code. Finally, a full-fledged composition language will be defined.

Acknowledgements. The authors are partially supported by EU Grant IST-33710: Aspect-Oriented, Model-Driven Product Line Engineering (AMPLE).

References

[1] P. Clements and L. M. Northrop, *Software Product Lines: Practices and Patterns*. Boston, USA: Addison-Wesley, 2002.
[2] D. M. Weiss and C. T. R. Lai, *Software Product-line Engineering: a Family-based Software Development Process*. Boston, USA: Addison-Wesley, 1999.
[3] K. Pohl, et al, *Software Product Line Engineering: Foundations, Principles and Techniques*. Berlin, Germany: Springer, 2005.
[4] K. Czarnecki and U. W. Eisenecker, *Generative Programming: Methods, Tools, and Applications*: ACM Press/Addison-Wesley, 2000.
[5] K. Kang, et al, "Feature-Oriented Domain Analysis (FODA) Feasibility Study", SEI, CMU/SEI-90-TR-021, 1990.
[6] K. Czarnecki and M. Antkiewicz, "Mapping Features to Models: A Template Approach Based on Superimposed Variants", presented at GPCE, Tallinn, Estonia, 2005.
[7] A. Bragança and R. J. Machado, "Automating Mappings between Use Case Diagrams and Feature Models for Software Product Lines", presented at SPLC, Kyoto, Japan, 2007.
[8] H. Gomaa, *Designing Software Product Lines with UML: From Use Cases to Pattern-Based Software Architectures*: Addison-Wesley, 2004.
[9] M. L. Griss, et al, "Integrating Feature Modeling with the RSEB", presented at ICSR, 1998.
[10] J. Bézivin, "On the Unification Power of Models", *Software and Systems Modeling*, vol. 4(2), pp. 171-188, 2005.
[11] L. Chung, et al, *Non-Functional Requirements in Software Engineering*, 1 ed: Kluwer Academic Publishers, 1999.
[12] A. Sampaio, et al "EA-Miner: A Tool for Automating Aspect-Oriented Requirements Identification", in *Proceedings of ASE*, Long Beach, CA, USA, ACM Press, 2005, pp. 352-355.
[13] K. Chen, et al, "An Approach to Constructing Feature Models Based on Requirements Clustering", in *Proceedings of RE*, Paris, France, IEEE Computer Society, 2005, pp. 31-40.
[14] A. Moreira, J. Araújo, and J. Whittle, "Modeling Volatile Concerns as Aspects", presented at CAiSE, Luxemburg, Luxemburg, 2006.
[15] "AMPLE Project Research Group at FCT/UNL", http://ample.di.fct.unl.pt/.
[16] P. Kruchten, *The Rational Unified Process: An Introduction*: Addison-Wesley, 2003.
[17] AMPLE, "Ample Project", http://www.ample-project.net/.
[18] M. Alférez, et al, "Traceability between Features and UML-Based Requirements Models: A Model-Driven Approach for Product Lines Engineering", http://ample.di.fct.unl.pt/tool/Alferez-etal-TR-1-2008.pdf
[19] N. Maiden and I. Alexander, *Scenarios, Stories, Use Cases: Through the Systems Development Life-Cycle*: John Wiley & Sons, 2004.

9

Multi-View Composition Language for Software Product Line Requirements

Authors: Mauricio Alférez, João Santos, Ana Moreira, Alessandro Garcia, Uirá Kulesza, João Araújo, Vasco Amaral.

Paper Summary: This paper proposes the Variability Modeling Language for Requirements (VML4RE), a requirements composition language for SPLs. VML4RE has two main goals: (i) to support the definition of relationships between SPL features expressed in feature models and requirements expressed in multiple models; and (ii) to specify the composition of requirements models for deriving specific SPL products using a simple set of operators. This paper details the composition language and refines the composition activity described in Appendix 8.

Authors Contribution: Mauricio Alférez was the main author and responsible for the main part of the research and writing of this paper, accounting for the 90% of the work. Other authors gave interesting comments that helped to improve the content of the paper.

Publication Arena: Published in the book "Software Language Engineering, Second International Conference, SLE 2009, Denver, CO, USA, October 5-6, 2009, Revised Selected Papers". Acceptance rate: 19%. Conference classification: CORE B. [92, 15].

Multi-View Composition Language for Software Product Line Requirements

Mauricio Alférez[1], João Santos[1], Ana Moreira[1],
Alessandro Garcia[2], Uirá Kulesza[1], João Araújo[1], Vasco Amaral[1]

[1] New University of Lisbon, Caparica, Portugal
[2] Pontifical Catholic University of Rio de Janeiro, Brazil

{mauricio.alferez, joao.santos, amm, uira, ja, vasco.amaral}@di.fct.unl.pt
afgarcia@inf.puc-rio.br

Abstract. Composition of requirements models in Software Product Line (SPL) development enables stakeholders to derive the requirements of target software products and, very important, to reason about them. Given the growing complexity of SPL development and the various stakeholders involved, their requirements are often specified from heterogeneous, partial views. However, existing requirements composition languages are very limited to generate specific requirements views for SPL products. They do not provide specialized composition rules for referencing and composing elements in recurring requirements models, such as use cases and activity models. This paper presents a multi-view composition language for SPL requirements, the *Variability Modeling Language for Requirements* (VML4RE). This language describes how requirements elements expressed in different models should be composed to generate a specific SPL product. The use of VML4RE is illustrated with UML-based requirements models defined for a home automation SPL case study. The language is evaluated with additional case studies from different application domains, such as mobile phones and sales management.

Keywords: Requirements Engineering, Software Product Lines, Variability Management, Composition Languages, Requirements Reuse.

1 Introduction

Software Product Lines (SPLs) engineering [1, 2] is an increasingly-relevant approach to improve software quality and productivity. It encompasses the creation and management of systems' families for a particular domain. Each system (product) in the family is derived from a shared set of core assets. Thus, a SPL product shares many features with the other products. SPL features are typically represented in domain analysis using feature models [3, 4]. A feature [3] is a visible system property or functionality that is relevant to some stakeholders. Features are either commonalities or variabilities used to distinguish products of an SPL. A feature

model is used to capture such commonalities and variabilities of the products' family in a SPL, and define their dependencies.

Model-based development methods for SPLs [2, 5, 6] support the construction of different models to provide a better understanding of each SPL feature. However, features, which are modeled separately in partial views, must be composed to show the requirements of the target applications. Composing variable and common requirements is a challenging task. Requirements are the early software artifacts most frequently documented in a multi-view fashion. Their description is typically based on significantly heterogeneous languages, such as use cases [7] (coarse-grained operational view), interaction diagrams (fine-grained operational view) [8], goal models (intentional and quality view) [9, 10], and natural language. This varied list of requirements models is a direct consequence of requirements having to be understood by stakeholders with different backgrounds, from customers of specific products to SPL architects, programmers and testers.

However, initial work on compositional approaches [2, 5, 6, 11] for requirements artifacts is rather limited in language support. These approaches do not offer composition operators for combining common and varying requirements based on different partial views. They are also often of limited scope and expressiveness [11]. Therefore, a key problem in SPL remains to be addressed*: how to compose elements defined in separated and heterogeneous requirements models using a simple set of operators?*

This paper answers this question by proposing the *Variability Modeling Language for Requirements* (VML4RE), a requirements composition language for SPLs. VML4RE has two main goals: (i) to support the definition of relationships between SPL features expressed in feature models and requirements expressed in multiple models; and (ii) to specify the composition of requirements models for deriving specific SPL products using a simple set of operators.

VML4RE provides a set of specialized operators for referencing and composing requirements elements of specific types, based on recognizable abstractions used in the domain of each requirements modeling notation or technique. Such operators can help SPL engineers to understand and choose the composition rules for requirements models. In contrast with conventional and general-purpose languages for model transformation, such as XTend [12], ATL [13] and AGG [14], VML4RE is tailored to requirements composition in a way that is accessible to requirements engineers. This is an important contribution of our work, as it addresses the problem of abstraction mismatch caused by such general-purpose model transformation languages [15, 16]. VML4RE prevent SPL designers from the burden of language intricacies that are not part of the abstraction level at which they are used to work.

The remainder of this paper is organized as follows. Section 2 presents a set of criteria used when creating the requirements variability composition language. Section 3 describes a case study that is later used to illustrate the VML4RE composition language and creates an example specification. Section 4 presents VML4RE and Section 5 discusses its application to the case study. Section 6 presents the evaluation of the language and discusses its benefits and limitations. Section 7 examines related work and compares it with ours. Finally, Section 8 concludes the paper and points directions to future work.

2 Criteria to Design VML4RE

SPL Requirements Engineering handles both common and variable requirements that enable the derivation of customized products of the family. Feature models are used to specify SPL commonalities and variabilities and feature model configurations are used as a driver during the process to derive product-specific requirements models. *Requirements variability composition* is the ability to customize requirements models for specific family products. The customization of model-based requirements implies a composition process where some elements are added, others are removed and, eventually, some are modified from the initial models. This section describes five criteria taken into account for the design of the VML4RE. These criteria raised from the needs for requirements models specification and composition for heterogeneous SPLs proposed by the industrial partners in the AMPLE project [17]:

C1: Support Multi-View Variability Composition: Requirements are the early software artifacts most recurrently documented in a multi-view fashion. In this context, variability manifests itself in different kinds of requirements (e.g., functional requirements and quality attributes) and design constraints (e.g., different databases, network types or operating systems) [2]. Modeling the requirements using multiple views facilitates the understanding of the SPL's variabilities and its specific products. This is particularly important in SPL development as it encompasses a number of stakeholders, from customers and managers of specific products, to core SPL architects, programmers and testers.

C2: Provide Requirements-Specific Composition Operators: Requirements descriptions are typically based on significantly-heterogeneous languages. Specific composition operators for combining common and varying requirements based on elements used in different views or models facilitate the operators' adoption by the SPL developers. General-purpose composition languages, such as XTend [12], ATL [13] and AGG [14], require a deep knowledge of the abstract syntax of the models to describe their composition. This highlights the problem of abstraction mismatch and the need for a composition language that does not require additional developer expertise. Requirements engineers should work at the same level of abstraction they are used to [15].

C3: Support Fine- and Coarse-Grained Composition: Requirements models can represent different levels of detail for a specific product. Coarse-grained modeling helps to define the scope of the system to be built by expressing the goals or the main functions of the product. Each coarse-grained element is often associated with a variety of fine-grained elements. The latter provide detailed requirements for what the system must do or sub-goals of the different parts of the system. For instance, UML provides coarse-grained model elements, such as packages and use cases, to organize the main subsystems and functions of the system to be built. Then, other models, such as activity diagrams, support further refinements of use cases. As a result, both fine-grained and coarse-grained composition is required to address the different levels of abstraction employed in SPL requirements engineering.

C4: Support Positive and Negative Variability: In general, there are three means to derive models for a specific SPL product: positive variability, negative variability and a combination of both. *Negative* variability is the removal of optional elements from a given structure, whereas *positive* variability is the addition of optional parts to

a given core [18]. Optional elements are related to optional and alternative features of the SPL and the core part encompasses features that are common to all the products. Sanchez et al. [15] presented a positive-negative modeling technique for variability management, but the composition operators are specific to architectural models. The flexibility provided by a positive-negative approach for composition is also advisable for requirements models. For example, in cases where the addition of a model element requires the removal of other(s) elements, as often happens when modeling mutually-exclusive features.

C5: Facilitate Trace Links Generation: Variability specification usually keeps implicit information governing the relationships between each SPL feature and respective requirements models. Composition methods could support explicit traceability of varying features through the generation of trace links from variability specifications. Hence, traceability information could be used to analyze system evolution properties, such as change impact analysis or requirements coverage.

The five criteria described above formed the basis for the VML4RE design. The use of the VML4RE language assumes a process workflow, which is described in Figure 1. *Domain Engineering* encompasses the creation of a set of artifacts associated with the SPL. These artifacts are reused in application engineering to produce specific SPL products. VML4RE is useful at the first stage of domain engineering, called *domain analysis*. *Variability identification* and *SPL requirements modeling* are the most important activities, which are performed in parallel during domain analysis. During *variability identification* (Figure 1-A), distinction is made between core (common) SPL features and the features of specific products. SPL r*equirements modeling* (Figure 1-B) tackles the detailed specification of features using different requirements modeling techniques and notations (related to *C1*). *Composition specification* (Figure 1-C) relies on requirements-specific composition rules to specify how to customize requirements models (related to *C2*). These rules can be based on operators that address both fine- and coarse grained compositions (related to *C3*).

The reusable artifacts created in domain engineering are used in application engineering to derive specific product models through the definition of configurations. Existing product derivation tools like pure::variants [19] and Gears [20] mainly allow to derive the complete or partial source code of a product. The input to this derivation is the existing code artifacts produced for a SPL architecture. However, these tools do not provide language support for the derivation of requirements models for a specific product (related to *C2*). In a VML4RE-centric process, *variability resolution* (Figure 1-D) implies selection of the variable features to be included in the product. Finally, *models derivation* (Figure 1-E) is the actual composition of the different models of a specific product. This supports the addition and removal of elements from the initial models (related to *C4*). Additionally, when deriving the models, appropriate tool support can be able to generate the trace links (Figure 1-F) between the features chosen for the product and the different parts of the requirements models (related to *C5*). This paper focuses on the *Composition Specification* activity highlighted in grey in Figure 1. The next section presents the case study and introduces VML4RE as a way of addressing the five criteria just discussed.

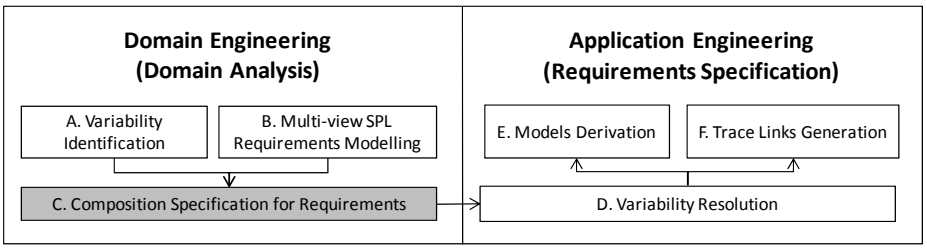


Fig. 1. Key Steps of SPL Requirements Composition

3 Case Study: Home Automation

Smart Home is a software product line for home automation being developed by Siemens AG [21]. For brevity and clarity, we rely on a subset of the Smart Home features. The left hand side of Figure 2 shows the partial feature model of the product line, while the middle of the figure presents one of its possible configurations, the "Economical Smart Home" (to create the models we use the FMP tool [22]). Some optional features are not included in such an Economical edition. Therefore, camera surveillance and internet as user interfaces are not part of the final product, for example. Hence, these features are not ticked in the product feature model (middle).

Figure 3 presents the use case model of the Economical Home as an exemplar of the set of models that we intend to obtain after the composition process. The elements highlighted in grey are related to variable features selected to be included in the Economical Home, while the rest of the elements are related to common features. Table 2 gives an example of the relationships between features and parts of the models. Also, the following sections provide more details on how this model was composed.

Smart Home inhabitants must be able to adjust the heater of the house to their preferred value (*Manual Heating* feature). In addition, the *Smart Heating* feature might be activated in a house. If so, a heater control will adjust itself automatically to save energy. For instance, the daily schedule of each inhabitant is stored in the Smart Home gateway. When the house is empty, the heater is turned off and later turned back on, on time to reach the desired temperature when the inhabitants return home. Smart Home can also choose to open or close windows automatically, to regulate the temperature inside the house as an option to save energy (*Electronic Windows* feature). Alternatively to the electronic windows, the inhabitants could always be able to open and close the windows manually (*Manual Windows* feature).

There are different types of graphical user interfaces that allow monitoring and managing the different devices of the smart home as well as receive security notifications (*GUI* feature). The available GUIs alternatives are: touch screens inside the house (*Touch Screen* feature), or via internet through a website and a notifier (*Internet* feature). As far as the *Security* feature is concerned, inhabitants can initiate the secure mode by activating the glass break sensors or/and camera surveillance devices (*Glass Break Sensors* and *Cameras* features). If an alarm signal is sent by any of these devices, and according to the security configuration of the house, the Smart

Home decides to (i) send a notification to the security company and the inhabitants via internet and touch screens, (ii) Secure the house by activating the alarms (*Siren* and *Lights* features), and/or (iii) closing windows and doors (*Electronic Windows* feature). Next we introduce VML4RE and illustrate its use with this case study.

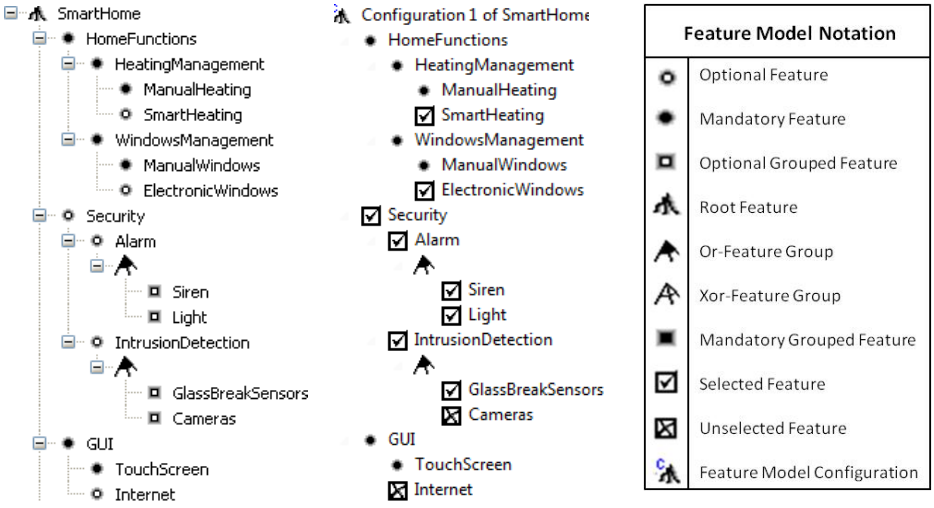


Fig. 2. (left) Smart Home Feature Model; (middle) Feature Model Configuration for the Economic Home; (right) Feature Model Notation

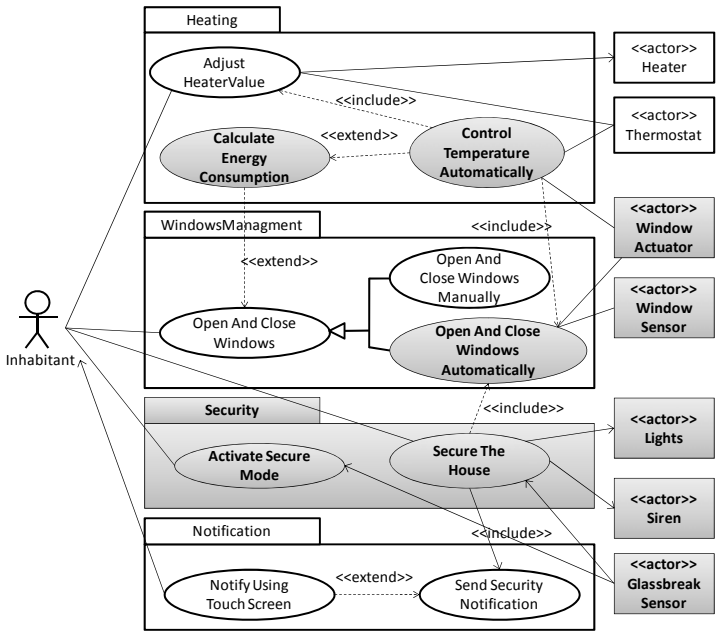


Fig. 3. Smart Home Economical Edition Use Case Model

4 VML4RE

This section outlines the VML4RE process, its main elements and its composition semantics.

4.1 VML4RE Process

The VML4RE process is described by instantiating the requirements composition process outlined in Figure 1. Figure 4 shows the specific artifacts employed in each of the activities. For *variability identification* (Figure 4-A), we employ a feature model that specifies the common and variable features of the SPL, as well as their dependencies. For *requirements modeling*, we employ various requirements models. In particular, we chose *use cases* whose detailed behavior is modeled using *activity models*. This mimics what often happens in mainstream UML-based methods, such as RUP [23]. The further elaboration of use cases with activity models; in contrast to free-format textual descriptions, facilitate the adoption of model-driven generation tools. This alternative provides models that conform to a metamodel (i.e., the metamodel of UML activity diagrams), thereby reducing the ambiguity in the specifications [2]. The detailed specification of use cases as activity models also enables customizations of use cases realizing specific SPL configurations.

During requirements modeling, other models, such as goal models [9, 10], can be used to specify interactions between functional and non-functional requirements. Such models also allow studying the actors and their dependencies, thus encouraging a deeper understanding of the business process. In addition, goal models can be used as a way to introduce intentionality in the elicitation and analysis of requirements. As a consequence, these goals allow the underlying rationale to be exploited in the selection of variants in the application development process [24].

The VML4RE specification (Figure 4-C) references the requirements models and specifies composition rules (also called *actions*). The VML4RE interpreter (Figure 4-E and F) receives as input the SPL REquirements (RE) models (Figure 4-B), the feature model configuration (Figure 4-D) and the VML4RE specification (Figure 4-C). As output, the interpreter generates: (i) use cases of a product; (ii) activity models that describe product usage scenarios; (iii) additional requirements models, such as, goal models (Figure 4-E); and (iv) the trace links between features and specific elements in the requirements models (Figure 4-F).

4.2 VML4RE Main Elements

Each VML4RE specification is composed of three main kinds of elements:

1. *Importing*: it imports the set of requirements and feature model that are used in the VML4RE specification. This is accomplished using *import* sentences.
2. *Commonalities*: it defines the features that are mandatory to every product of a SPL. It is used to reference the parts of the requirements that are related to SPL *common* features.

3. *Variabilities:* it defines the variable (*optional*, *variation points and variants*) features of the SPL. *Optional* features are not mandatory and might not be included in some of the products of the SPL. A *variation point* identifies a particular concept within the SPL requirements specification as being variable and it offers a number of *variants*. A *variant* describes a particular variability decision, such as a specific choice among alternative variants. The *variabilities* blocks are used to: (i) reference (sentences initiated by the keyword *ref*) the requirements related to each variable feature, and (ii) enclose operators used to compose the requirements related to each variable feature.

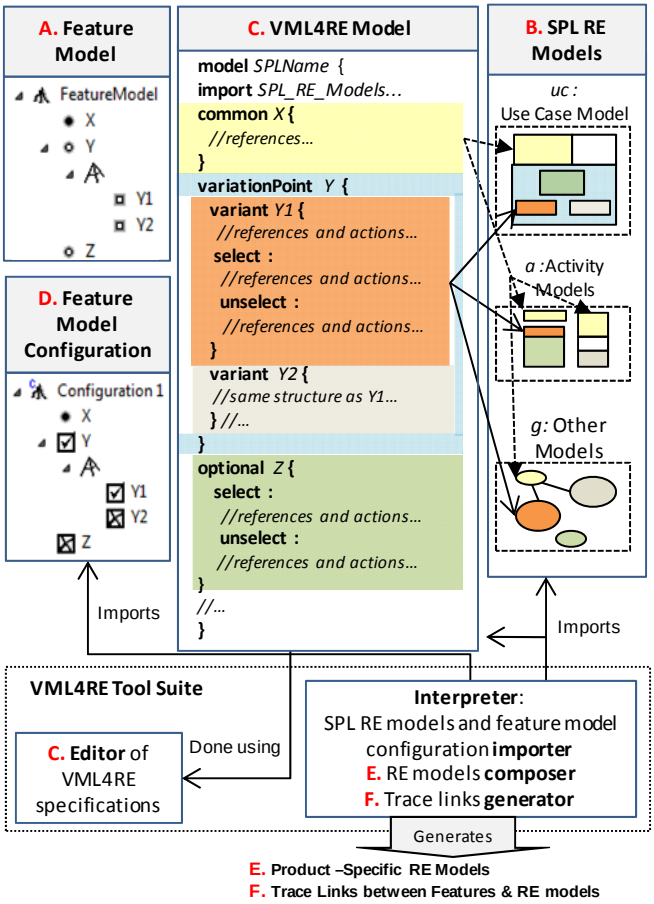


Fig. 4. Artifacts and Composition Workflow

The VML4RE specification outline (in Figure 4-C) contains separated blocks for import sentences, common features like X, and variable features like Y, Y1, Y2 and Z. Each *optional*, *variationPoint* and *variant* blocks can have *select* and *unselect* sub-blocks. They indicate the set of references and actions that are taken into account if the feature was selected or not in the feature model configuration. Thus, given that Y

and Y1 are selected in the feature model configuration, the actions and references inside the *select* block of feature Y1 are executed. The actions and references inside the *unselect* block of the Y2 and Z features are also executed.

4.3 References and Composition Operators

VML4RE provides *references* to indicate which elements in the requirements models are related to specific features. Also, it provides a set of specialized operators for composing requirements model *element*s like *use cases*, *packages*, *activities* or *goals*.

The upper part of Table 1 summarizes the description of the structure of the elements related with the references. In VML4RE specifications, the *ref* statements allow creating a reference between the different common, optional and alternative features and specific parts of models. In the *ref* statements, it is possible to use designators (e.g., ".", "equal") and quantification (e.g., "*" that indicates all the elements inside a model element). Logical operators like "And" and "Or" can be used to create more complex query expressions over the models. Listing 1 provides examples of references to packages, activities and use cases that will be explained in the description of the Smart Home section. Table 1 also summarizes the structure of some composition operators. These include operators that are relevant to use case, activity and goal models (in particular, the strategic dependency model of the *i** [10] goal-oriented approach). Analogous to the *insert* operators that add parts to the base model, we have *replace* and *remove* operators. The complete metamodel and grammar of the language can be found in [25].

The semantics of each VML4RE composition operator can be defined in terms of a model-to-model transformation. For instance, the "Insert Use Case Links" operator using the use case link type "*associatedWith*", connects an actor and a use case using an association link (for example, `insert(UCLinks_of_type: associatedWith{from actorD to useCaseModelA.PackageB.useCaseC});`). The intended transformation of the use case model can be presented by the left hand side (LHS) and right hand side (RHS) graphs in Figure 5, where the inputs are a use case model, a use case, a use case's package, and an actor. If there is already an association between the actor and the use case in the same package, the transformation is not applied to avoid duplicates. This is expressed with the cross in some elements in the LHS graph that act as negative application conditions (NAC). It means that any match against the LHS graph cannot have a *packageB* with any existing association between *actorD* and the *useCaseC*.

In general, a graph transformation is a graph rule $r: L \rightarrowtail R$ for LHS graph L to a RHS graph R. The process of applying r to a graph G involves finding a graph monomorphism, h, from L to G and replacing $h(L)$ in G with $h(R)$ [26]. The notation used to define this graph transformation is similar to the one used by [27] where the LHS and RHS patterns are denoted by a generalized form of object diagrams. However, for visual simplicity we added dashed lines between elements to represent any number of containments (in this case, package's containments). We defer to [27] for the readers interested in details of this notation.

Figure 6 illustrates the replace operator with the example "Replace use case". A *replace* in this context includes to *remove* a use case and then insert a new use case

linked in the place of the old use case (for example, `replace (useCase useCaseModelA.useCaseB by UseCase useCaseC } ) ; )`.

Table 1. Some of the VML4RE elements.

Element	Description and Structure of Some Elements Related with References

Reference	Identifies one or more requirements model elements. The references are made to specific types of elements in the models. This is expressed using the designator *ofType* that allows querying based on the type of model element (*ElementType*), e.g., *UseCase*, *Activity*, *Actor*, *or Element* when the referenced models elements are of different types.
	`Reference : "ref" ref_name ofType ElementType"{"(RefExpression | ref_name2) WhereDeclaration?"}";`
	`RefExpression : elementName (("." RefExpression)|".*")?;`

Where Declaration	It is an optional part of a reference expression that allows querying a set of model elements based on their name. `WhereDeclaration : "Where" "(" Expression ")";`

Expression	Some of the possible *designators* are*: equal*, *different*, *startsBy*, *finishesWith*, and *contains*. They search for matches between a *literal* and the first letters, last letters or in any place in the names of the model elements of a specific type, respectively. Besides, the expressions can be combined with logical *operators* like *and* and *or* to create more complex queries.
	`Expression : BooleanExpression(SubExpression)*;`
	`SubExpression : Operator BooleanExpression ;`
	`BooleanExpression : "contains" literal | "equal" literal | "different" literal | "startsBy" literal | "finishesWith" literal`

Element	Description and Structure of Some Actions
Insert Package	Insertion of a package in a use case model, or in another package `"package" package_name "into" RefExpression;`
Insert Use Case	Insertion of a use case into a use case model or inside a package(s) `"useCase" useCase_name "into" RefExpression;`
Insert Use Case Links	Insertion of different relationships between elements in a use case model `"UClinks_of_type:" UseCaseLinkType "{" UCElementsLinkage+ "}" ;`
UC Elements Linkage	Helps to factorize the insertion of relationships in a use case model (*Insert Use Case Links*) according to the *UseCaseLinkType* for a better organization of the actions. `"from " RefExpression "to" RefExpression ("," RefExpression )*;`
Use Case Link Type	Available relationships between use cases (*inherits*, *extends*, *includes*) and between actors and use cases (*associatedWith* and *biAssociatedWith* for bidirectional relationships). `("inherits" | "extends" | "includes" | "associatedWith" | "biAssociatedWith");`
Insert Actor	Insertion of an actor into a use case model or package `"actor" actorName "into" RefExpression ;`
Insert Activity	Inserts an activity into an activity model `"activity" (newActivityName "into" RefExpression) ;`
Activity Elements Flow	Helps to factorize the insertion of relationships in an activity model (*InsertActivityLinks*, not shown in this table) and optionally to add a guard condition. `"from" RefExpression "to" RefExpression ("with guard" guardCondition)?;`
Replace Use Case	Replaces a use case by a new one `"useCase" RefExpression "by" "useCase" newUseCaseName;`
Replace Activity	Replaces an activity by a new activity or a complete activity model. `"activity" RefExpression "by"(("activity" newActivityName)|("activityModel" RefExpression);`
Insert iGoal	Inserts a goal of *I* (indicated by the i in iGoal)* in a strategic dependency model. "iGoal" *goalName* `"into" RefExpression ;`
Insert iGoal dependencies	Insertion of different dependencies relationships between elements in a strategic dependency model. `"IGoalDependencies_of_type:" iGoalDependencyType "{" iGoalElementsLinkage+ "}" ;`
iGoal Elements Linkage	The links between the nodes in the strategic dependency diagram go from *depender* to dependee through *dependum*. `"from " RefExpression "to" RefExpression "through" dependumName;`
iGoal Dependency Type	`("resourceDependency"|"taskDependency"|"goalDependency"|"SoftGoalDependency");`

The advantage of specifying model compositions with a pure graph transformation approach is its expressivity by allowing accessing all the elements of the metamodel. However, software modelers typically do not have the in-depth knowledge about intricacies of the requirements metamodels required to specify a graph rule [28]. The *actions* in VML4RE do not require any kind of knowledge about the details of the

metamodels. They provide requirements-specific composition operators that facilitate the specification of the composition of the models.

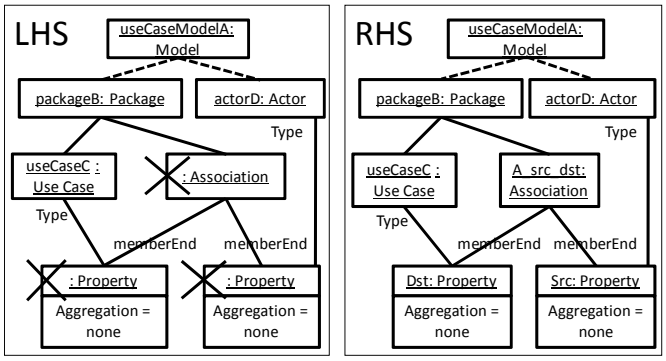


Fig. 5. Graph Rule to Insert an Association between ActorC and useCaseB in PackageB

Fig. 6. Graph Rule to Replace UseCaseB by UseCaseC

5 Applying VML4RE

This section illustrates the use of the references and some VML4RE actions for domain and application engineering.

5.1 VML4RE in Domain Engineering

The Smart Home requirements were modeled with use case and activity models created with the UML Tools plug-in [29]. The FMP tool [22] was used to build a feature model to specify SPL commonalities and variabilities. This tool supports cardinality-based feature models. The relations between the models are specified with VML4RE. The VML4RE editor is implemented using xText [30], a framework for the development of textual DSLs. It is part of the VML4RE tool suite [25] implemented in the Eclipse platform as a set of extensible plugins. It is based on openArchitectureWare [12], a model-driven development infra-structure, and the Eclipse Modeling Framework (EMF) [31]. Listing 1 shows a partial view of this specification. Initially, the different requirements and feature models are imported to be used in the specification (Lines 2-4).

In the VML4RE specification, the modeler can create references to requirements models. For instance, it is possible to reference a specific element in a model, like an actor; this happens in "ref Heater ofType Actor {uc.Heater}" (line 10); or all the

elements (e.g., use cases, packages, actors) inside one container element, e.g., "ref AllHeatingElementsInUCs ofType Element{uc.Heating.*}" (lines 13-15); or elements in different parts of the models according to a search condition, like "ref SurDev ofType Activity {ams.* Where equal VerifyInstalledSurveillanceDevice} " that searches in the set of activity models for activities with the name "VerifyInstalledSurveillanceDevice" (lines 51-52).

```
01  model SmartHome {
02  import UseCaseModel '/UCModel.uml' as uc;
03  import FeatureModel '/FeatModel.fmp' as fm;
04  import ActivityModels '/ActModels.uml' as ams;
05  /* more imports sentences...*/
06  Common HomeFunctions { //...
07    Common HeatingManagement {//...
08      select:
09      ref Heating ofType Package {uc.Heating}
10      ref Heater ofType Actor {uc.Heater} //...
11      Common ManualHeating { /*...*/}
12      Optional SmartHeating { /*...*/}
13        ref AllHeatingElementsInUCs
14            ofType Element  { uc.Heating.* }
15      }
16    Common WindowsManagement {
17      select:
18      ref WindowsMngmt ofType Package
19                { uc.WindowsManagement }
20      Common ManualWindows { /*...*/ }
21      Optional ElectronicWindows { /*...*/ }
22    }
23    ref AllWindowsMngmtElements ofType
24       Element { uc.WindowsManagement.* }
25  }
26  Optional Security {//...
27    select:
28    insert ( package Security into uc );
29    ref Security_Pkg ofType
30          Package { uc.Security }
31    Insert (useCase SecureTheHouse
32              into Security_Pkg);
33    ref SecHouse ofType UseCase
34              { Security_Pkg.SecureTheHouse }
35    insert ( useCase ActivateSecureMode
36              into Security_Pkg ) ;
37    ref ActSecMode ofType UseCase
38          { Security_Pkg.ActivateSecureMode }
39    VariationPoint Alarm { //...
40      Variant Siren { /*...*/ }
41      Variant Light { /*...*/ }
42    }
43    VariationPoint IntrusionDetection { //...
44      Variant GlassBreakSensors { //...
45        select :
46        insert ( actor GlassBreakSensor into uc ) ;
47        ref GlassSen_A ofType Actor
48                          { uc.GlassBreakSensor }
49        insert ( UClinks_of_type: associatedWith
50              { from GlassSen_A to SecHouse } );
51        ref SurDev ofType Activity {ams.* Where
52            equal VerifyInstalledSurveillanceDevice }
53        replace ( SurDev by Activity
54                  VerifyInstalledGlassBreakSensors );
55      }
56      Variant Cameras {
57      select :
58        insert ( actor Cameras into uc );
59        ref Camera_A ofType Actor {uc.Cameras}//..
60      }
61    }
62  }
63    Common GUI { /*...*/ }
64  }
```

Listing 1. VML4RE Model for the Smart Home

The VML4RE specification also employs *actions* to specify how variable requirements model elements are composed with common requirements model elements. Listing 1 presents several *actions* to be applied in activity and use case models. For example, the insertion of the *Security* package into the use case *uc* (line 28), or the insertion of the *SecureTheHouse* use case in the *Security* package (line 31-32) and the insertion of an association between the *GlassbreakSensor* actor and the use case *SecureTheHouse* (lines 49-50).

5.2 VML4RE in Application Engineering

In application engineering, the feature model configuration is used as a driver during the process to derive product-specific requirements models. Figure 2 (middle) shows the feature model configuration of an Economical Smart Home. The Economical Smart Home does not have camera surveillance or use internet to send security notifications. The VML4RE interpreter processes the SPL requirements models and the feature model configuration to derive a product-specific requirements model. During this process, we can use a positive, negative, or a mixture of positive and negative variability transformation strategies. Our interpreter first includes all the model requirements elements related to mandatory features by processing the respective *ref* statements specified inside of the *common* feature blocks. These elements are also called the *core model* in our approach, since they are included in every SPL instance. After that, the interpreter processes the *ref* statements and *actions* of the *variabilities.* In this example of the Smart Home, we will illustrate the use of the VML4RE in conjunction with a positive variability approach since we mostly used actions that add optional parts to the base model.

Finally, product-specific requirements models are produced based on processing the VML4RE specification (Listing 1). Given the possibility of defining, in a unique VML4RE specification, the relationships between a feature model and several requirements models (e.g., use case and activity models), our interpreter produces different product-specific requirements models in just one-step. Our current implementation [25] supports the derivation of use case and activity models, and we are working to address other models (*i** [10] and KAOS [32], for example).

During the Economical Smart Home derivation process, the actions and references related to *Internet* and *Cameras* were not included; for instance, the reference and action related to the *Cameras* actor (Listing 1, lines 58-59). The result of the composition of the use case model is shown in Figure 3, where the elements added to the core model were highlighted in grey. In addition to the use case model, other requirements models were transformed according to the execution of VML4RE actions. Figure 7 shows the *ActivateSecureMode* activity model, related to the use case with the same name. When the *Security* optional feature is chosen, the actions contained in the "*select"* block of the *Security* variation point are performed (Listing 1, lines 28-38), and the *ActivateSecureMode* (Figure 7 (left)) activity model is included into the final requirements product models.

During derivation of product-specific requirements models, some of the generic activities in the activity model can be replaced with others more specific to the product that is being configured. This happens in the Economical Home where the *GlassBreakSensors* are the only surveillance devices selected in the configuration. Hence, we could create a simple replacement of the *VerifyInstalledSurveillanceDevices* activity by the *VerifyInstalledGlassBreakSensors* activity as appears in Listing 1 (lines 47-48). As there are probably other activities for verification of surveillance devices in the requirements models, we use the *Where* operator. The result of the replacement is shown in Figure 7 (right). If two (or more) variants from an OR feature are selected, such as the *Intrusion Detection*, our interpreter produces two (or more) different activity models, one for each instance.

This strategy was developed to avoid conflicts in the transformation of a same activity model.

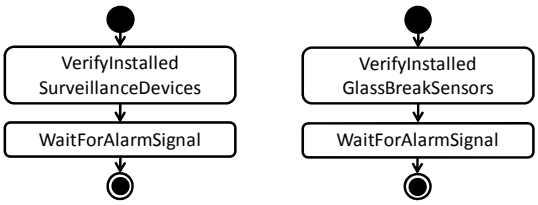


Fig. 7. Simplified Smart Home ActivateSecureMode Before and After a Replace Activity Action

VML4RE allows the product derivation of trace links between the features and elements in other models, such as use cases and activity diagrams. This derivation can be accomplished based on the *ref* sentences inside each of the common and variable blocks. Each *ref* in the VML4RE specification can determine several references between model elements from the feature model and the SPL requirements models. There may be also cases when an element in a requirements model is referenced by more than one feature. VML4RE specifications are processed automatically by our tool [25, 33] to generate all the set of links involving SPL requirements models (see Section 2, C5). Table 2 presents a partial set of the trace links relevant to the feature "Heating Management". These links are created based on references, such as the ones in Listing 1, lines 9-10, 13-14. Lines 9-10 refer to the package "Heating" and the actor "Heater", and lines 13-14 refer to any kind of element inside the "Heating" use case package like the use case "Control Temperature Automatically" and "Adjust Heater Value".

Table 2. Part of the Trace Links Generated by the References in the Heating Management Feature.

Feature	Element	
	Name	Type
Heating Management	Heating	Package
	Heater	Actor
	Control Temperature Automatically	UseCase
	Adjust Heater Value	UseCase
	Smart Heating	ActivityModel
	…	…

6 Evaluation and Discussion

This section discusses the benefits and limitations of VML4RE based on our experience from the application of the language. We have evaluated the usefulness of VML4RE in three case studies [21], two of them proposed by partners of the European AMPLE project [17], the Smart Home proposed by Siemens A.G. [34], and a slice of the customer relationship management system, developed by SAP A.G [35] . The third case study is a product line for handling mobile media [36]. These three product lines are from different domains and exhibit different kinds of variability (e.g., options and variants). All of them encompassed textual requirements. Feature models and UML use cases were available for the Mobile Media and Sale Scenario

and an activity model was also available for the latter. The activity models of Mobile Media and Smart Home were translated from informal textual use case scenarios to activity models. The output models were validated by the original developers of the case studies. The goal models for the Sales Scenario system were produced by two teams of postgraduate students at Universidade Nova de Lisboa, based on the use case scenarios and market requirements provided by SAP A.G.

We evaluate the VML4RE usefulness based on the criteria for requirements models composition defined in Section 2 and then we discuss on additional benefits and limitations of VML4RE.

C1: Support Multi-view Variability Composition: Each feature block in VML4RE concentrates the *actions* related only with itself and that can transform models in multiple views of the requirements. VML4RE was initially designed to support the composition of two of the most commonly used requirements modeling techniques, such as use cases and activity models that address coarse and fine grained operational views of the requirements. We have also started using it with very different kinds of requirements modeling, like the goal-oriented modeling technique *i** [10], that addresses a quality and intentionality view of the requirements, as happened in the case of the Sales Scenario.

C2: Provide Requirements-Specific Composition Operators: as presented in Table 1, VML4RE provides specialized operators for composing requirements model elements of specific types, such as use cases, packages, activities or goals. The composition operators are simple and did not require from the modeler a deep knowledge on the relationships between the metamodel's metaclasses. For instance, the UML2.0 metamodel for the use cases has metaclasses like *property*, *association* and *classifier*. These metaclasses are important on the design of the transformations, but they are not needed when writing compositions with VML4RE. The composition description was simple in the three case studies because it was based on a vocabulary used in the domain of each modeling technique (e.g., *use case, associatedWith, package, dependency*).

C3: Support Fine and Coarse-Grained Composition: in the three case studies the coarse-grained composition was performed in terms of broadly-scoped elements, such as packages, use cases. The operators "*remove package*" and "*insert…use case*" are examples of such cases. VML4RE also addresses fine-grained composition when the *actions* are performed within coarse grained elements. The operators "*insert activity*", "insert activity links" are examples of such cases.

C4: Facilitate *Trace Links Generation:* As explained in Section 5.3, our approach supports the derivation of trace links. These links record relations between features specified in feature models and other requirements model elements pertaining to the SPL or to a specific product. This is accomplished with the *reference* sentences that are processed by the tool suite. We are currently exploring different traceability scenarios that process these relationships to expose useful information. This information could be exploited in many activities, such as discovering candidates of bad feature interactions and visualizing variations in different requirements models. Many of these traceability functionalities also facilitate the job of SPL architects as they are also valuable to analyze the design change impact when evolving SPL features and requirements.

C5: Support Positive and Negative Variability: VML4RE offers operators to support positive variability (e.g., *insert*) and negative variability through *remove* or *replace* operators. Positive variability presents some advantages for modeling and composing requirements models. For example, requirements modeling is characterized by the incremental knowledge acquisition about the system. In this sense, starting with a relatively small and easy to understand set of models seems to be a good starting point. Then, while the developer knows more about each feature of the SPL, s/he can incrementally specify how each new variable feature will modify the existing models. Positive variability also allows the management of variability in time. If the core model is created using generic requirements then, the requirements models are more flexible to include future specific requirements that instantiate the generic ones. Take for example Figure 7 (Right), it specifies that, at some point, it is necessary to "verify installed Surveillance devices". Then this requirement not only allows its instantiation for current surveillance devices, like "GlassBreak Sensors" and "Cameras", but it can also be instantiated by other unknown surveillance devices that were not initially considered in the SPL.

While modeling with VML4RE we saw additional benefits of the composition of requirements models. For example:

Testing and Understanding the Behavior of Specific Products: The automatic derivation of requirements models for a specific product is useful both to understand which requirements and features are involved in the development of an SPL product, and to support the testing and documentation activities. In particular, activity models are an example of requirements artifacts that are well suited for business process modeling and for modeling the logic captured by a single use case or scenario as happened during the modeling of the Sales Scenario. Activity models can provide a base to understand and validate the behavior of parts of a product of the SPL in the presence or absence of specific features. Also, using the goal-based modeling in the Sales Scenario allowed us to understand the dependencies between the actors, thus encouraging a deeper understanding of the business process.

Consistency Checking between Feature Models and other Requirements Models: Modeling different models in large systems like SPLs can be a difficult, time-consuming, and highly error-prone task, if appropriate supporting tools are not available. During the realization of our three case studies, we noticed that the generated trace links from VML4RE specifications, can be process by our traceability framework [33] to detect inconsistencies between features and requirements in different models. Examples of such inconsistencies are: (i) inexistence of features related to specific requirements; (ii) inexistence of requirements related to specific features; (iii) conflicts between features acting over the same requirements (which can be valid or not). The consistency management of the relationships between features and requirements models is also fundamental to help the functionalities' tracing mentioned above, especially in SPL evolution scenarios.

Finally, we came across with some issues during the application of the compositions. When creating the composition actions for each variation point with the core model, the modeler could have assumptions regarding the existence, position and name of the model elements. However, the models change after the application of each insertion, replacement or deletion of model elements and it could unable the application of some subsequent actions. It is necessary to determine the best

precedence order for the application of the actions in each variation point and also for the application of each variation point after model modifications. Existing formal methods and model-checking techniques and tools like simulation, or critical pair analysis as introduced by Javaraman, et al.[16] may be the first solution candidates.

7 Related Work

Most of the work on feature composition is focused on implementation, such as Ahead [37] and pure::variants [19]. There are a couple of languages that focus on the architecture level like VML4Architecture [38] and Koala [39]. Recently, some approaches were proposed to support the definition of relationships between SPL features and requirements and the composition of requirements models. Pohl [2] separates variability from functional information, in an orthogonal model. He proposes a variability metamodel that includes the following two relationships: the Artifact Dependency relationship (that relates variants with development artifacts), and VP Artifact Dependency (that relates variation points to development artifacts). These elements enable the definition of links between the variability model and other development artifacts. Nevertheless, this work is focused on documenting variability, rather than expressing how to specify the composition of requirements models.

Czarnecki and Antkiewicz [6], and Bragança and Machado [40] create explicit relationships between features and requirements. Czarnecki and Antkiewicz propose a general template-based approach which enables the creation of relationships between elements (abstractions and relationships) from an existing model to the corresponding features through a set of annotations. The annotations are used mainly to indicate the presence of conditions of specific model elements or model templates according to feature occurrences. In contrast with VML4RE that allows positive, negative or positive-negative variability, Czarnecki and Antkiewicz [6] only employ negative variability. Bragança and Machado use a simplified feature model based on the one proposed by Czarnecki and Antkiewicz and employ UML notes in use case diagrams to indicate variability. These notes are linked to *includes* and *extends* relationships, providing variability data. The main disadvantage with these two approaches [6, 40] is that they fail to fully separate functional and variability information as they use intrusive graphical elements such as, presence conditions or notes in their models to indicate variability. Hence, variability information may be scattered and polluting the models, making them difficult to understand and maintain.

Gomaa [5] extends UML-based modeling methods for single systems to address software product lines. He uses stereotypes (e.g., <<kernel>>, <<optional>> or <<alternative>>) to indicate variability, models use case packages as features in a feature model, and manually relates features with other model elements using matrixes. Variability stereotypes and other kinds of stereotypes are mixed in the same models reducing the understandability of the models.

Although the previous approaches provide techniques to establish the relationships between feature and requirements models, they lack a language to specify the actual composition of different requirements models. Our work proposes a requirements

specific language and tool support to deal with composition of requirements models for software product lines.

There are other approaches that provide languages to create reference expressions and composition rules. XWeaver [18], for example, supports the composition of different architectural viewpoints. It composes crosscutting concerns encapsulated as aspect models into non-aspect-oriented base models, thus following an asymmetric composition approach (though this could be extended for symmetric approach with relatively little effort). XWeaver is similar to our approach because the composition is done based on matching names of elements in the aspect and the base model. It employs an OCL-like expressions language [12] that play the role of the VML4RE's references. However, it does not provide requirements specific composition operators.

MATA [28] is an aspect-oriented approach to model composition based on graph rewriting formalisms that can be used to compose models of different SPL features to create models specific of products [16]. It employs graphical patterns that resemble the concrete syntax of specific kinds of UML models (e.g., state machines). In aspect-oriented terminology, graphical patterns can be thought of as pointcuts and the composition operators can be thought of as the advices. Similarly, in VML4RE, *references* can be thought of as the pointcuts and the *actions* as the advices. VML4RE, in comparison to MATA, provides more simple operators that are especially tailored to facilitate writing composition of requirements models. However, VML4RE can complement MATA by providing concrete language support to express in the same code block of each feature, the references and composition rules for all the different requirements views. VML4RE together with similar variability composition languages focused on architecture like VML4Architecture [38] could be used as an alternative frontend for MATA.

Apel et al. [11] employ superimposition of feature-related model fragments as a general models' composition technique. We believe that this technique can be especially useful in requirements to compose coarse-grained models that keep a common structure in a positive-variability setting. However, to be more useful in a broader kind of requirements models, it requires language support to express also positive-negative variability, and to reference potentially multiple composition points for model fragments during fine-grained composition.

8 Conclusions and Future Work

VML4RE address the question on *how to compose elements defined in separated and heterogeneous requirements models using a simple set of operators*. It was designed taking into account the five fundamental criteria discussed in Section 2. Section 6 reviewed how these criteria are addressed. VML4RE presents a contribution to the field of language support for composing SPL requirements due to its unique characteristics: (1) each feature block (e.g., common, variant) concentrates a cohesive set of *actions* that can transform models in multiple requirements views; (2) new composition operators are especially tailored for canonical requirements models and rely on a vocabulary familiar to requirements engineers; (3) there is an explicit separation between the modeling of variability and requirements, without forcing the

intrusive inclusion of variability-related elements in requirements models; (4) the new operators that can *add, remove* or *replace* parts of the models, thus supporting both positive and negative variability; (5) the use of references to facilitate the creation of compositions and the generation of trace links.

Currently, we are investigating the application of model-driven techniques to keep consistent the relationships between SPL variability and requirements models during models' evolution. Also, we are studying an effective way to determine the best precedence order for the application of the actions in each variation point and also for the application of each variation point after model modifications. Finally, we are interested in showing the use of our language using other requirements views and improving the usability of VML4RE by including a graphical concrete syntax.

Acknowledgments. This work is supported by the European FP7 STREP project AMPLE [17].

References

1. Clements, P., Northrop, L.M.: Software Product Lines: Practices and Patterns. Addison-Wesley, Boston, MA, USA (2002)
2. Pohl, K., Böckle, G., van der Linden, F.: Software Product Line Engineering: Foundations, Principles and Techniques. Springer, Berlin, Germany (2005)
3. Czarnecki, K., Eisenecker, U.W.: Generative Programming: Methods, Tools, and Applications. ACM Press/Addison-Wesley Publishing Co. (2000)
4. Kang, K., Cohen, S., Hess, J., Novak, W., Peterson, A.: Feature-Oriented Domain Analysis (FODA) Feasibility Study. Technical Report CMU/SEI-90-TR-021, Software Engineering Institute, Carnegie Mellon University (1990)
5. Gomaa, H.: Designing Software Product Lines with UML: From Use Cases to Pattern-Based Software Architectures. Addison-Wesley (2004)
6. Czarnecki, K., Antkiewicz, M.: Mapping Features to Models: A Template Approach Based on Superimposed Variants. In: Glueck, R., Lowry, M.R. (eds.): GPCE'05, Vol. 3676. Springer, Tallinn, Estonia (2005) 422-437
7. Alexander, I., Maiden, N.: Scenarios, Stories, Use Cases. Wiley, Chichester, UK (2004)
8. Unified Modeling Language (UML) Superstructure , Version 2.1.2 : 2007-11-02,
9. Chung, L., Nixon, B., Yu, E., Mylopoulos, J.: Non-Functional Requirements in Software Engineering. Kluwer Academic Publishers (1999)
10. i* an Agent-oriented Modelling Framework http://www.cs.toronto.edu/km/istar/
11. Apel, S., Janda, F., Trujillo, S., Kästner, C.: Model Superimposition in Software Product Lines. ICMT'09, Zurich, Switzerland (2009)
12. openArchitectureWare, http://www.openarchitectureware.org/
13. Jouault, F., Kurtev, I.: Transforming Models with ATL. In: Bruel, J.-M. (ed.): Model Transformations in Practice Workshop at MoDELS'05, Vol. 3844. Springer, Montego Bay, Jamaica (2005) 128-138
14. Taentzer, G.: AGG: A Graph Transformation Environment for Modeling and Validation of Software.: 2nd International Workshop on Applications of Graph Transformations with Industrial Relevance (AGTIVE), Virginia, USA (2003)
15. Sánchez, P., Loughran, N., Fuentes, L., Garcia, A.: Engineering Languages for Specifying Product-derivation Processes in Software Product Lines. SLE'08, Toulouse, France (2008)

16. Jayaraman, P., Whittle, J., Elkhodary, A., Gomaa, H.: Model Composition in Product Lines and Feature Interaction Detection Using Critical Pair Analysis. MODELS'07, Vol. 4735. Springer, Nashville, USA (2007) 151-165
17. Ample Project, http://www.ample-project.net/
18. Groher, I., Volter, M.: XWeave: Models and Aspects in Concert. 10th International Workshop on Aspect-oriented Modeling. ACM, Vancouver, Canada (2007)
19. pure::variants, http://www.pure-systems.com/Variant_Management.49.0.html
20. Gears, http://www.biglever.com/
21. Morganho, H., Gomes, C., Pimentão, J.P., Ribeiro, R., Grammel, B., Pohl, C., Rummler, A., Schwanninger, C., Fiege, L., Jaeger, M.: Requirement Specifications for Industrial Case Studies. Technical Report, D5.2, AMPLE Project (2008)
22. Antkiewicz, M., Czarnecki, K.: FeaturePlugin: Feature Modeling Plug-in for Eclipse. 2004 OOPSLA workshop on eclipse technology eXchange. ACM Press, Vancouver, British Columbia, Canada (2004) 67-72
23. Kruchten, P.: The Rational Unified Process: An Introduction. Addison-Wesley Longman Publishing Co., Inc. (2003)
24. González-Baixauli, B., Laguna, M.A., Leite, J.C.S.d.P.: Using Goal-Models to Analyze Variability. Variability Modelling of Software-intensive Systems, Limerick, Ireland (2007)
25. Variability Modeling Language for Requirements, http://ample.di.fct.unl.pt/VML_4_RE/
26. Grzegorz, R. (ed.): Handbook of Graph Grammars and Computing by Graph Transformation: Volume I. Foundations. World Scientific Publishing Co., Inc., River Edge, NJ, USA (1997)
27. Markovic, S., Baar, T.: Refactoring OCL Annotated UML Class Diagram. MODELS05, Motengo Bay (2005)
28. Whittle, J., Moreira, A., Araújo, J., Jayaraman, P., Elkhodary, A., Rabbi, R.: An Expressive Aspect Composition Language for UML State Diagrams. In: G. Engels, e.a. (ed.): ACM/IEEE MODELS'07, Vol. 4735. Springer, Nashville, TN, USA (2007) 514–528
29. MDT-UML2Tools, http://www.eclipse.org/uml2/
30. Xtext Reference Documentation, http://www.eclipse.org/gmt/oaw/doc/4.1/r80_xtextReference.pdf
31. Eclipse Modeling Framework, http://www.eclipse.org/modeling/emf/?project=emf
32. Goal-Driven Requirements Engineering: the KAOS Approach, http://www.info.ucl.ac.be/~avl/ReqEng.html
33. Sousa, A., Kulesza, U., Rummler, A., Anquetil, N., Mitschke, R., Moreira, A., Amaral, V., Araújo, J.: A Model-Driven Traceability Framework to Software Product Line Development. 4th Traceability Workshop held in conjunction with ECMDA, Berlin, Germany (2008)
34. Siemens AG - Research & Development, w1.siemens.com/innovation/en/index.php
35. SAP A.G, www.sap.com/about/company/research/centers/dresden.epx
36. Figueiredo, E., Cacho, N., Sant'Anna, C., Monteiro, M., Kulesza, U., Garcia, A., Soares, S., Ferrari, F.C., Khan, S., Filho, F.C., Dantas, F.: Evolving software product lines with aspects: an empirical study on design stability. ICSE'08. ACM, Leipzig, Germany (2008)
37. AHEAD Tool Suite, www.cs.utexas.edu/users/schwartz/ATS.html
38. Loughran, N., Sánchez, P., Garcia, A., Fuentes, L.: Language Support for Managing Variability in Architectural Models. 7th International Symposium on Software Composition, Vol. 4954. Springer, Budapest, Hungary (2008), pp. 36-51
39. Rob van, O., Frank van der, L., Jeff, K., Jeff, M.: The Koala Component Model for Consumer Electronics Software. Computer, vol. 33, 3, pp. 78--85. IEEE Comp. Society (2000)
40. Bragança, A., Machado, R.J.: Automating Mappings between Use Case Diagrams and Feature Models for Software Product Lines. SPLC, Vol. 0., pp. 3-12, IEEE Computer Society, Kyoto, Japan (2007)

10

VML* – A Family of Languages for Variability Management in Software Product Lines

Authors: Steffen Zschaler, Pablo Sánchez, João Santos, Mauricio Alférez, Awais Rashid, Lidia Fuentes, Ana Moreira, João Araújo, Uirá Kulesza.

Paper Summary: The key contribution of this paper is in the domain of software-language engineering, where it applies ideas from SPL engineering and model-driven development to the development of VML* languages. This enables developers to efficiently build new VML* languages for new SPL contexts avoiding error-proneness of language development. A secondary contribution is that this new approach to language development supports additional capacities for VML* languages, such as generation of trace links or syntax verification. The capacity of VML* to use composed expressions instead of only atomic expressions was implemented in new versions of VML4RE. That capacity helped us to improve the VML4RE version described in Chapter 9 to finally obtain the version described in Section 3.4 - Inside VML4RE.

Authors Contribution: Steffen Zschaler was the main author and responsible of the development of tool support and the main part of the writing of this paper accounting for the 60% of the work approximately. Research was the result of the generalization of VML4Arch and VML4RE languages that before the creation of VML* were created in isolation but sharing important characteristics. Contributions regarding VML4Arch were mostly provided by Pablo Sánchez and Lidia Fuentes, and for VML4RE by João Santos and Mauricio Alférez. Ana Moreira, Awais Rashid, João Araújo, and Uirá Kulesza gave

interesting comments that helped to improve the content of the paper.

Publication Arena: Published in the book "Software Language Engineering, Second International Conference, SLE 2009, Denver, CO, USA, October 5-6, 2009, Revised Selected Papers". Acceptance rate: 19%. Conference classification CORE B.[13]

VML* – A Family of Languages for Variability Management in Software Product Lines[1]

Steffen Zschaler[1], Pablo Sánchez[2], João Santos[3], Mauricio Alférez[3], Awais Rashid[1], Lidia Fuentes[2], Ana Moreira[3], João Araújo[3], Uirá Kulesza[3]
[1] Computing Department, Lancaster University, Lancaster, United Kingdom
{zschaler, awais}@comp.lancs.ac.uk
[2] Dpto. de Lenguajes y Ciencias de la Computación, Universidad de Málaga, Málaga, Spain
{pablo,lff}@lcc.uma.es
[3] Computer Science Department, Universidade Nova de Lisboa, Lisbon, Portugal
{jps, mauricio.alferez, amm, ja}@di.fct.unl.pt, uirakulesza@gmail.com

Abstract. Managing variability is a challenging issue in software-product-line engineering. A key part of variability management is the ability to express explicitly the relationship between variability models (expressing the variability in the problem space, for example using feature models) and other artefacts of the product line, for example, requirements models and architecture models. Once these relations have been made explicit, they can be used for a number of purposes, most importantly for product derivation, but also for the generation of trace links or for checking the consistency of a product-line architecture. This paper bootstraps techniques from product-line engineering to produce a family of languages for variability management for easing the creation of new members of the family of languages. We show that developing such language families is feasible and demonstrate the flexibility of our language family by applying it to the development of two variability-management languages.

Keywords: Software Product Lines, Family of Languages, Domain-specific Languages, Variability Management.

1 Introduction

Software Product Lines Engineering (SPLE) is seen as a promising approach to increasing the productivity and quality of software, especially where essentially similar software needs to be provided for a variety of contexts and customers each requiring customizations and variations for their specific conditions [1-2]. In SPLE, features [3] are used to capture commonalities or discriminate among products, i.e. capture variabilities, in an SPL. SPL features are often modelled using feature models [3-4]. Management of variability throughout the product line is a key challenge in SPLE.

An important part of variability management is to make explicit the relation between the variability model (e.g., the feature models referred to in the previous para-

[1] The work reported in this paper was supported by the EC FP7 STREP project AMPLE: Aspect-Oriented Model-Driven Product Line Engineering (www.ample-project.net).

graph) and other models and artefacts of the SPL. Once this relation has been explicitly represented, it can be used for a number of purposes, most importantly to automatically derive product instances based on product-configuration specifications, but also for other purposes such as trace-link generation and consistency checking of SPL models. Due to its relevance, this topic is currently an area of intensive research and a number of approaches have been proposed [5-9]. Initial research focused on using general-purpose model transformations to encode product derivation [10-11]. Later it was argued that this placed too heavy a burden on SPL engineers, as they would now also have to learn the intricacies of model transformations. Consequently, a number of approaches that hide the model transformations from the SPL engineers have recently been developed [6-7, 12].

Czarnecki et al and Heidenreich et al [6-7] propose generic techniques that associate features with arbitrary combinations of model elements and generate a standard model transformation for product derivation from this. In contrast, we have argued before [12] [13] that transformation actions that are specific to the types of models used for describing the SPL are more useful, as they provide a terminology already known to SPL engineers, allow consideration of model semantics in the definition of transformations, and allow avoiding some inconsistencies (e.g., dangling references) in product models by design.

This requires new languages to be developed for each type of model that may be used in describing an SPL—a costly and error prone task. To make development of such languages feasible, this paper proposes VML*[2], a family of languages—or a language product line—for variability management, showing that developing such languages is a feasible goal. Individual members of the family are described using a domain-specific language (DSL). Based on such a specification, a generator produces the complete infrastructure for the specified language. Such a generative approach has the added benefit of making it easier to support other evaluations beyond product derivation: they can be implemented in additional code generators from the language specification.

The key contribution of this paper is, thus, in the domain of software-language engineering, where it applies ideas from SPLE and model-driven development to the development of VML* languages. This enables us to efficiently build new VML* languages for new SPL contexts, and thus improves over our previous work [12], which was limited to copy-and-paste-based reuse, limiting efficiency and increasing error-proneness of language development. A secondary contribution is that this new approach to language development allows us to support additional evaluations for VML* languages, such as generation of trace links or SPL consistency checking.

Section 2 further discusses the motivation for building custom languages instead of one generic language and derives a set of challenges to be overcome to enable efficient development of such languages. Section 3 then presents how we applied SPLE techniques to construct a family of languages for variability management and is followed by Sect. 4, which shows how concrete languages have been developed based on our approach. Section 5 reviews some related work and Sect. 6 concludes the paper and points out directions for future work.

[2] For Variability Management Languages

2 Motivation

This section describes the motivation that led to the creation of the VML* family of languages. First, we provide some background on VML languages and then we present the motivation of this paper.

2.1 Managing Variability Using Target-Model–Specific Languages

This section explains why we choose to model SPL variability using target-model–specific languages rather than a single generic language. We use as an example an architectural model of a lock control framework for a Smart Home Software Product Line (SPL) [1, 14]. Smart Home applications aim at automating and controlling houses and buildings in order to improve the comfort and security of their inhabitants. The lock control is placed on doors of rooms whose access must be controlled. Several options are available to end users acquiring a specific Smart Home software installation:

- Different authentication mechanisms can be used: identification cards, fingerprint scanners or a simple numeric keypad.
- Doors are opened manually and users have a time period to authenticate before triggering the alarms. Optionally, it is possible to select a computer-controlled door lock control (Automatic Lock), which will be released upon successful authentication.
- Automatic sliding doors can also be used (Door Opener). This option requires that the Automatic Lock control of the door lock be selected.

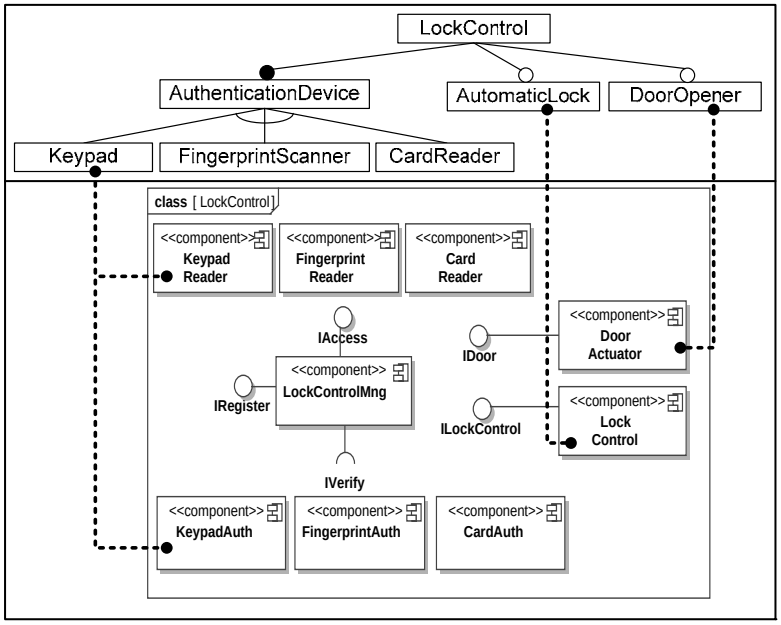


Figure 1 A software architecture for the lock control framework

Figure 1 depicts a software architectural design for this lock control framework. This architectural design is comprised of three different parts, which are explained in the following.

Firstly, variability inherent to the domain is expressed using a feature model [4, 15] (Fig. 1 (top)). This feature model represents variability specification or problem space. It specifies which features of the system are variable and the reasons why. For instance, the `AuthenticationDevice` to be used is a variable feature because there are several alternative devices available but only one must be selected. `Automatic-Lock` and `DoorOpener` are variable features because they are options that may be included in a specific lock control application or not.

Secondly, once variability has been identified, the software architecture is designed using the component model of UML 2.0 (Fig. 1 (bottom)). This represents variability realization or solution space. The mechanism selected for supporting variability in the architectural design is plugin components. The `LockControlMng` component is the central component of this architecture. Each alternative for authentication is designed as a pair of plugin components: one for controlling the physical device that serves to authenticate users (e.g. `KeypadReader`); and the other one encapsulating the logic of the authentication algorithm (e.g. `KeypadAuth`). These plugin components communicate with the `LockControlMng` through the `IAccess` interface, in the case of reader components, and the `IVerify` interface, in the case of authenticator ones. All plugin components must register in the `LockControlMng` component using the interface `IRegister`. The `LockControlMng` receives data from the reader components and, with the data received, it calls the authenticator component. The latter is in charge of checking if the user has access to the room or not. If the user is authentic, the `Lock-ControlMng` component invokes the `LockControl` component, which releases the lock. This invocation is placed only if the automatic lock control option has been selected. If the door is a sliding one, the `LockControlMng` should also invoke the `DoorActuator` component for automatic opening of the door.

Thirdly, we must specify the links between variability specification and variability design, or problem space and solution space, indicating how the components of the architectural model must be composed according to the selected features. In our case, for instance, when a specific authentication device is selected, the corresponding reader component must be connected to the `LockControlMng` through the `IAccess` interface. In the same way, the `LockControlMng` component must be connected, to the corresponding authenticator component though the `IVerify` interface. Both the authenticator and the reader components must also be connected to `LockCon-trolMng` through the `IRegister` interface. The components corresponding to non selected alternatives must simply be removed. Similarly, the `DoorActuator` and `LockControl` components are adequately connected if the corresponding optional features are selected; otherwise, they should be removed.

These relationships can be expressed using general purpose model transformation languages, such as demonstrated in [10-11]. Nevertheless, as previously discussed in [10], these have the following shortcomings:

- *Metamodel Burden.* A model transformation language is often based on abstract syntax manipulations. According to Jayaraman et al. [16], “Most model

Table 1 Part of the VML4Arch Specification for Smart Home

```
01  import features </SmartHome.fmp>;
02  import core </SmartHome.uml>;
03
04  ...
05
06  variant for FingerprintScanner {
07      connect("FingerprintReader","LockControlMng","IAccess");
08      connect("FingerprintReader","LockControlMng","IRegister");
09      connect("FingerprintAuth","LockControlMng","IRegister");
10      connect("LockControlMng","FingerprintAuth","IVerify");
11  } // Fingerprint scanner
12
13  variant for not (FingerprintScanner) {
14          remove('FingerprintReader');
15          remove('FingerprintAuth');
16  } // not FingerprintScanner
```

developers do not have this knowledge. Therefore, it would be inadvisable to force them to use the abstract syntax of the models".

- *Language Overload and Abstraction Mismatch.* There are different kinds of model transformation languages [16], and each of them is based on a specific computing model. They range from rule-based languages (e.g. ATL [17]) to expression-based languages (e.g. xTend [18]) and graph-based languages (e.g. AGG [19]). When employing a model transformation language, software product line engineers must also understand the underlying computing style (e.g. rule-based) and learn the language syntax. As a result, software product line engineers are forced to rely on abstractions that might not be naturally part of the abstraction level at which they are working.

To overcome these shortcomings, we proposed [12] to create dedicated languages, for specifying product derivation processes; that is, for specifying how features map to software models. These dedicated languages must follow a very basic computation style, where based on a selection of features, small sequence of simple commands are executed. These commands, moreover, must use syntax familiar to the modeler, using concepts of the concrete syntax of the model rather than their abstract syntax. These user-friendly high-level specifications are then translated into a set of low-level general purpose model transformations, which support the automation of the product derivation process. So, the SPL engineer can enjoy the benefits of using model-driven techniques but without paying the associated cost, i.e. without needing to learn the intricacies of model transformation languages. Table 1 provides an example of such a dedicated language for manipulating UML component models.

This specification establishes that whenever the `Fingerprint` option is selected (lines 06-11), the `KeypadAuth` and `KeypadReader` components must be connected to the `LockControlMng` component through the corresponding interfaces, as previously described. The `connect` operator is an intuitive composition mechanism to specify that two components must be connected using the interface specified as a parameter. The first parameter of the connect operator is the component that requires the interface while the second parameter is the component that provides it. In the case where the `Fingerprint` variant is not selected (lines 03-16), the `FingerprintAuth`

and the `KeypadReader` components are removed from the architecture, using the `remove` operator.

2.2 Automating the generation of new VML languages

Beyond the language from Figure 1, a wide range of languages for managing variability in any kind of target modeling language need to be constructed. For instance, we need to develop a dedicated language with specific operators for managing variability in use cases models, activity models, business process models or any other kind of architectural description language. Developing such languages is cost-intensive and error-prone, especially as so far there is no support for reuse between different such languages beyond a copy-and-paste approach. This is a serious barrier to the adoption of our approach in SPL projects.
To make developing such languages feasible, we need to solve the following three challenges:

1. *Support of reuse between different languages.* The support infrastructure should be easily reused for new languages. Reuse should not be based on copying an existing language implementation and adjusting it, removing unneeded actions and adding new actions. Otherwise, if errors are found and fixed in the infrastructure for one language, these corrections would have to be manually transferred into all other language infrastructures. The same would be true for new features of the infrastructure, for example, new evaluations of specifications other than product derivation.
2. *Allow the type of variability models to vary.* Different approaches to modelling variability have been proposed: very often, feature trees [4] or cardinality-based feature models [20] are used. However, DSLs have also been used to represent variability [21]. Any variability management language should be easily adapted to any type of variability model.
3. *Support for easy customisation of target-model element access.* Target-model model elements need to be accessed from a specification based on a textual reference (e.g., their fully qualified name or some pattern matching a number of names). Depending on the target model different forms of such textual references may be useful. The evaluation of such textual references should be implemented separately from the individual actions to allow for easy exchange and customisation of this feature.

In this work, we present a generative infrastructure for creating new VML languages for a concrete target model that tackles these issues.

3 The VML* Family of Languages

In response to the challenges identified in the previous section, we propose to bootstrap SPLE techniques using a model-driven and generative approach for creating the infrastructure (e.g., parser, editor, evaluation engine) for a specific VML* language. To this end, we have developed the VML* family of languages, which consists of:

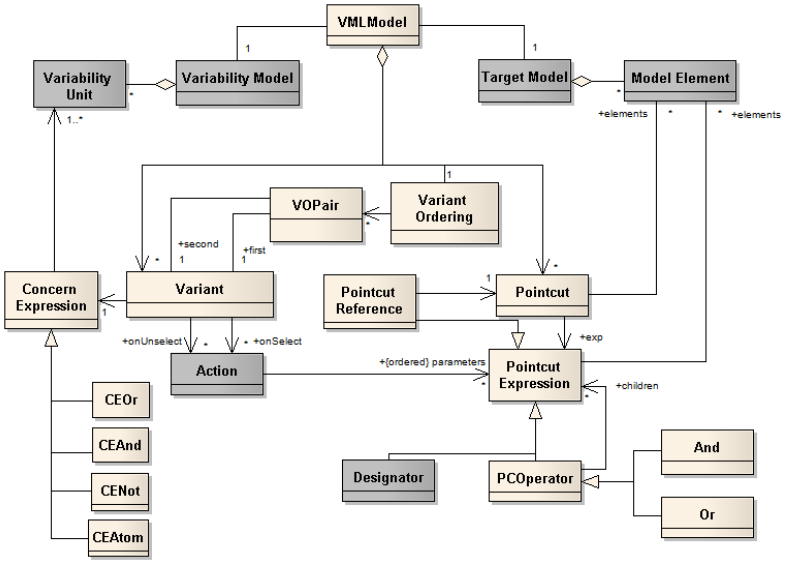


Figure 2 Common metamodel for VML languages. Variation points have been highlighted in dark grey

1. A common metamodel for VML* languages including variation points that can be customised for describing specific VML* languages. This provides the concepts common to all VML* languages.
2. A DSL for specifying the choices a specific language makes for each variation point.
3. A generator-based infrastructure that can instantiate all custom elements of the process from [12] for any VML* language.

A working prototype of this system is available as a set of Eclipse plugins [22].

3.1 A common metamodel for VML* languages

Figure 2 shows the general concepts required for expressing variability in product-line models. This metamodel has been developed as a generalisation of the metamodels of VML4Architecture, or simply VML4Arch [12-13] and VML4Requirements, or simply VML4RE [23-24], two variability management languages we have previously developed. VML4Arch is a language for relating feature models and UML2.0 architectural models of an SPL. VML4RE is a language for relating feature models and UML2.0 use case and activity models. These languages have been developed in parallel, but independently. They have a number of differences, but they also share a large number of commonalities, enabling us to derive a common metamodel for VML* languages.

The metamodel shown in Figure 2 is independent of both the specific models used for variability modelling (e.g., feature models, domain-specific languages) and the specific target models (e.g., UML, architecture description models, generation work-

flow models). Consequently, a number of concepts are abstract in this metamodel. To apply the metamodel for a specific combination of target model and variability model, these concepts (highlighted in dark grey in Figure 2) need to be specialised (how to specify such specialisations will be discussed in the next section). In the following, we discuss each of the metamodel concepts in more detail.

VMLModel. A VML model relates a variability model and a target model, using a set of variants to describe how the target model needs to vary as each of the concerns of the variability model is selected or unselected.

VariabilityModel. A variability model is the central artefact in variability modelling. `VariabilityModel` and `Variability Unit` serve as adapters to the specific form of variability modelling employed in a specific scenario.

Variability Unit. These are the units of variability in variability modelling. A variability model describes what variability units a potential product may have and what constraints govern the selection of combinations of variability units for individual products. From the perspective of variability management, we are mainly interested in the name of a variability unit and whether it has been selected for a specific product configuration. Notice that for the purposes of our metamodel we do not care about how variability units are expressed in a variability model. They may be represented as explicit features in a feature model [4] or more implicitly in a DSL [21], or in any other form that is convenient for modelling variability in a specific project. To enable our metamodel to relate to all these different kinds of representations, we standardise on the common notion of Variability Unit and require adapters that extract these from any of the representations discussed above.

TargetModel. Target models describe a product line. There are a large number of potential target models—for example, requirements models, architecture models, or code-generation-workflow models.

ModelElement. Model elements represent arbitrary elements of the target model. This concept serves as an adapter to actual model elements and needs to be specialized for each kind of target model (thereby defining the concrete model elements available). The model elements are typed using metaclasses imported from the target metamodel.

Variant. A variant describes how the target models must be varied when a certain combination of variability units is selected or unselected. Notice that for product derivation it is sufficient to provide a variant for each non-mandatory variability unit, as we can assume the unvaried target model to represent the model for all the mandatory variability units. For some other evaluations (e.g., trace-link generation), however, a variant must be provided for each variability unit including mandatory ones. Each variant defines two sets of actions for its variability units: a set of *onSelect* actions defines how to vary the target model when the variability units are selected; a set of *onUnSelect* actions defines what to do when the variability units are not selected.

ConcernExpression. For certain use cases it is not sufficient to map variability units directly onto modifications of the target model, as has also been previously discussed in the literature [6-7]. Therefore, we define variants for so-called concern expressions, logic expressions over variability units. We support *And*, *Or*, and *Not* expressions as well as atomic terms.

VariantOrdering. Sometimes the order in which the actions of different variants are executed during product derivation is important, as actions for one variant may

rely on model elements created by actions for another variant. VariantOrdering provides SPL developers with a means of defining a partial order of execution over variants using pairs of variants. The infrastructure will guarantee that all actions of the first variant in a pair are executed before any action of the second variant of that pair is executed.

Action. Actions are used to describe modifications to the target model. These need to be customised for each kind of target model, depending on the kinds of variations that make sense at the level of abstraction the target model covers. For example, if the target model is a use case model, one particular action may be to connect an actor and a use case, while for an architectural model a possible action could be to connect two components. Actions may add, update or remove model elements in the target model and may create, update or remove links between existing or newly added model elements.

PointcutExpression. A pointcut expression is an expression that identifies a model element or a set of model elements. It is constructed from atomic designators, pointcut references and combining operators (*Not*, *And*, and *Or*).

Pointcut. A pointcut identifies a model element or set of model elements. The model elements are denoted by a pointcut expression. The main purpose of the Pointcut concept is to allow particular pointcut expressions to be named. A named Pointcut can then be reused using a PointcutReference.

PCOperator. Operators enable the construction of pointcut expressions combining the set of elements returned from more than one element pointcut. Here, we define only two operators, namely *and* and *or*, which represent intersection and union of the sets of model elements of their element expressions, respectively.

Designator. A designator is a piece of text that is used to identify a model element or a set of model elements. It may be a name (possibly qualified), a signature, a wildcard expression, or anything else that makes sense in the target model. As resolution of designator text into actual model elements is specific to the target model, the designator concept needs to be customised for each target model.

3.2 A DSL for specifying individual VML* languages

To enable succinct description of the specificities of a certain VML* language, we have defined a metamodel and concrete syntax for language-instance description. Figure 3 shows the key concepts. Based on an instance of this metamodel—a VML* language description—we can then generate an appropriate infrastructure customised for that specific VML* language.

The individual concepts in the language-description metamodel are:

LanguageInstanceModel. The central metaclass of VML* language descriptors, binding together the other parts of a VML* language descriptor.

VariabilityModelImport. This provides information about the type of variability model to be supported by the VML* language. The key interface between VML* and a variability model is the set of features defined. The language descriptor, therefore,

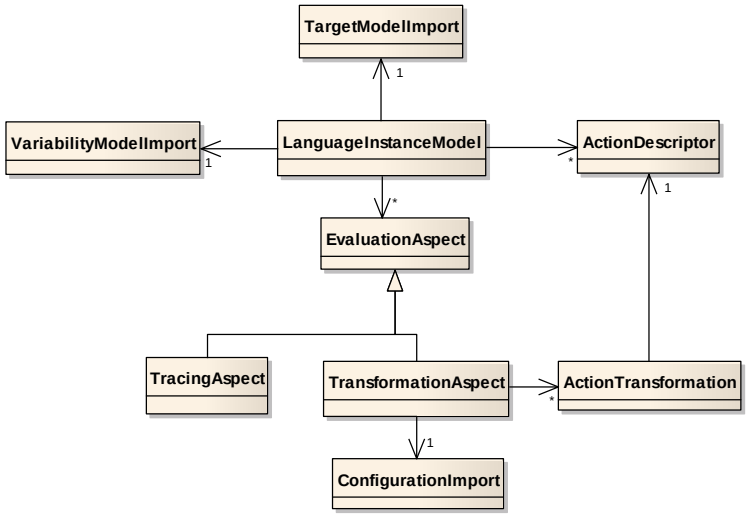


Figure 3 Metamodel for VML* language instance descriptions.

contains a snippet of model-query code[3] that serves as an adapter between the variability model and a VML* specification. This snippet is the only place where knowledge about the variability-model metamodel is located in a VML* language descriptor.

TargetModelImport. This provides information about the type of target model to be supported by the VML* language. Mainly, this defines how pointcut designators should be evaluated for a specific target model. Depending on the specific kind of target model, different pointcut designators may be required. While, for example, use-case models require only simple qualified names (possibly using wildcards for quantification) to identify individual actors, use cases, or activities, architectural models may additionally require pointcut designators for operation signatures or component provided or required interfaces. Therefore, both the syntax of pointcut designators and their interpretation is specific to the kind of target model. In all VML* languages, pointcut designators are syntactically represented as simple string values. They are then passed to a piece of model-query code interpreting them to return a set of model elements from a given target model. This piece of code is defined for a specific VML* language using TargetModelImport.

ActionDescriptor. Each action descriptor provides general syntactic information about one action. This includes the name of the action and the number of parameters it takes. The concrete syntax for action invocation in the generated VML* language will be '<action_name> ($param_1$, ..., $param_n$)'. For each parameter, users of the VML* language will be able to provide a pointcut expression.

[3] Our prototype uses openArchitectureWare's (oAW) xTend language to express model queries and model transformations. These xTend snippets can be kept as operations in a separate xTend file and referenced from the language instance descriptor, allowing language designers to take full advantage of oAW's checking capabilities.

EvaluationAspect. Every evaluation aspect describes one form of evaluation of a VML* specification. The VML* family can be extended with a number of these evaluation aspects (currently only one aspect—product derivation—has been implemented, but we are working on an implementation for trace-link generation and are planning to work on consistency evaluation), which can be supported for every concrete VML* language, but not all VML* languages will need support for all evaluation aspects. A VML* language description can, therefore, include only those evaluation aspects that are actually required for this VML* language, providing an additional opportunity for optimisation. Notice that making such a selection manually based on the architecture presented in the previous subsection can be very difficult, as the different evaluation aspects actually overlap in some elements of the architecture (for example, in plugin configuration files). The model-driven approach not only allows a selection of one aspect or another, it additionally allows this selection to be changed flexibly, even experimentally.

TransformationAspect. If present, it enables product-derivation for target models. For each `ActionDescriptor` this defines an `ActionTransformation` specifying the model transformation encapsulated by this action. Furthermore, a `ConfigurationImport` defines an adapter for configuration models.

ConfigurationImport. For the construction of models for specific products, the VML* infrastructure requires access to the set of features selected in a specific product configuration. To avoid polluting the VML* infrastructure with knowledge about the inner structure of product configurations, `ConfigurationImport` provides a snippet of model-query code that serves as an adapter to product-configuration specifications by extracting the set of selected features from a product configuration.

ActionTransformation. Provides additional information for an action pertaining to the transformation of target models by this action. For every `ActionDescriptor` there needs to be a corresponding `ActionTransformation` instance. In particular, this includes a snippet of model-transformation code that implements the action. In this code, the parameters can be referenced as '$param_1$' thru '$param_n$'. The type of each parameter is defined in the `ActionTransformation`.

TracingAspect. If present, it enables the generation of trace links from a VML specification. Such trace links connect selected features and added or removed model elements of the target model. The tracing aspect is specified by naming the model-transformation operations that create or remove model elements; wildcards may be used to provide these names. VML* will then generate an aspect for the model transformation that advises these operations and creates appropriate trace links using the AMPLE Tracing Framework (ATF) [25].

3.3 Generation of VML* language infrastructure

Instances of this metamodel can be defined using a textual concrete syntax. Table 2

Table 2 Excerpt from the language descriptor for VML4RE

```
01  vml instance vml4req { // Define a new language called vml4req
02     // This section defines the type of variability model and
03     // how to access it
04     features {
05        metamodel  "/bin/fmp.ecore"
06        // Extracts all variability units from a variability model
07        function   "getAllFeatures"
08     }
09
10     // This section defines the type of target model and how to
11     // access it
12     target model {
13        metamodel  "UML2"
14        type       "uml::Package" // Metamodel type of a model
15        // Function to interpret pointcut designators
16        function   "dereferenceElement"
17     }
18
19     // Importing plugins and external specifications
20     bundles: "unl.vml4req", "ca.uwaterloo.gp.fmp", ...
21     extensions: "unl::vml4req::library::umlUtil"
22
23     // Syntactical definition of available actions
24     actions:
25        createInclude {
26           params "List[uml::UseCase]" "List[uml::UseCase]"
27        }
28        insertUseCase {
29           params "String" "uml::Package"
30        }
31        ...
32
33     // Definition of available evaluation aspects
34     aspects:
35        transformation { // Evaluation for product derivation
36           // Defines adapter for product-configuration access
37           features {
38              type  "String"
39              function "getAllSelectedFeatures"
40           }
41           // Definition of the semantics of actions as
42           // model transformations
43           createInclude {
44              function  "createIncludes"
45           }
46           insertUseCase {
47              function  "createUseCase"
48           }
49           ...
50        }
51  }
```

shows an excerpt of the language descriptor for VML4RE (cf. Sect. 4). Mapping this concrete syntax to the abstract syntax discussed above is rather straightforward so that we will not discuss it in any more detail here. It is worth noting, though, that this language descriptor does not contain complicated model-transformation code; all that is specified are the names of some functions. These functions with the actual model-transformation code are contained in an external file[4], allowing standard editors and error highlighting to be used when writing the code. Including the fully qualified name of the external file in the list after the "extensions" keyword ensures that the extension can be accessed from all relevant places in the generated code. Similarly, the "bundles" keyword lists other plugins that should be made available to any generated plugins. Here we include the plugin project containing our extension and the FMP plugin [26] providing support for cardinality-based feature models.

Furthermore, we have developed a generator that takes language descriptors such as shown in Table 2 and generates a set of Eclipse plugins containing the infrastructure for this language. The operational prototype can be obtained from [27]. The code generated by this generator is based on the work previously presented in [12]. The generation is completely automatic; the only manual input provided by language developers is the language instance descriptor and the implementations of the actions provided in a separate file. The complete infrastructure for editing, compiling, and executing specifications of the new VML language is encapsulated in the generator and can, thus, be reused for each new language.

4 Example languages from the VML* family

We have re-implemented both VML4Arch and VML4RE based on our new infrastructure. As VML4Arch has already been discussed extensively in [12], here we will focus on VML4RE. For VML4Arch we will only give a brief discussion of what needed to be changed to make it compatible with VML*. Both implementations can be downloaded from [24].

4.1 VML4RE

Requirements are most recurrently documented in a multi-view fashion [28-29]. Their description is typically based on considerably heterogeneous languages, such as use cases, activity diagrams, goal models, and natural language. Initial work on compositional approaches for early development artefacts does not clearly define composition operators for combining common and varying requirements based on different views or models. Therefore, a key problem in SPLE remains how to specify and apply the composition of elements defined in separated and heterogeneous requirements models.

With the Variability Modelling Language for Requirements (VML4RE) [23] we propose an initial solution for this problem by introducing a new requirements composition language for SPLs. VML4RE is a textual language with two main goals:

[4] An oAW xTend file for our prototype.

Table 3 Selected VML4RE actions for Use Case Models

Action Signature	Description
insertUseCase (String name, Package p)	A new use case named name is inserted into package p.
insertPackage (String name, Package p)	A new package named name is inserted into package p.
createActorToUseCaseLink (List[Actor] actors, List[UseCase] usecases)	A new connection is created between each of the actors and each of the use cases.
createInclude (List[UseCase] source, List[UseCase] target)	A new <<include>> dependency is created between each of the source use cases and each of the target use cases.

(i) to support the definition of relations between SPL features expressed in feature models and requirements expressed in multiple views (based on a number of UML diagram types, such as use case diagrams and activity diagrams); and (ii) to specify the compositions of requirements models for specific products of a SPL. VML4RE supports composition operators for UML use cases and activity models. It has been applied to case studies in domains such as home automation [23] and Mobile Applications [30]. It has shown great flexibility to specify composition rules and references to different kinds of elements in heterogeneous requirements models. The results of these experiments are encouraging and comparable with other approaches that support semi-automatic generation of trace-links relationships and composition between model elements in SPLs.

Table 3 shows an overview of some of the available actions of the VML4RE language for use cases. A more complete list can be found in [23]. VML4RE provides another set of actions for activity models, which are not shown here due to space restrictions.

Table 2 shows an excerpt from the language descriptor for VML4RE. It has been defined to map from feature models expressed using the FMP metamodel [26] to UML2 use case and activity models. This is expressed in the two sections named 'features' and 'target model', respectively, which also reference the functions to adapt to the feature model and to dereference pointcut designators in the target model. The real dereferencing code is implemented in the extension referenced through the 'extensions' keyword. The full language descriptor also specifies a tracing aspect. This is not shown in Table 2 for lack of space.

Finally, Table 4 shows an excerpt of a VML4RE specification for the Smart Home case study [23]. Lines 7 to 20 show the additional use cases needed when the Security feature is selected in a product configuration. Notice the use of wildcards on Line 13 to select all use cases in a package. If, and what, wildcards are supported and how they are evaluated is defined in the dereferenceElement operation invoked from the language instance descriptor in Table 2 on Line 16. Further, notice the use of a slightly more complex pointcut expression on Lines 16 to 19 of Table 4. This pointcut

Table 4 Part of the VML4RE Specification for Smart Home

```
01  import features <"/SmartHome.fmp">;
02
03  import core <"/SmartHome.uml">;
04
05  ...
06
07  variant Security {
08
09       insertPackage ("Security", "");
10       insertUseCase ("SecureTheHouse", "Security");
11       insertUseCase ("ActivateSecureMode", "Security");
12       createActorToUseCaseLink (
13          "Inhabitant", "Security::.*");
14       createInclude (
15          "Security::SecureTheHouse",
16          or (
17            "Notification::SendSecurityNotification",
18            "WindowsManagement::" _
19            "OpenAndCloseWindowsAutomatically"));
20  }
```

expression results in a set of two use cases: Notification::SendSecurityNotification and WindowsManagement::OpenAndCloseWindowsAutomatically.

4.2 VML4Arch

Re-implementing VML4Arch based on the VML* infrastructure proved surprisingly easy. However, as any product line requires a certain amount of stream-lining between individual products to maximise reuse, there were some minor adjustments we had to make to fit VML4Arch into the family of languages. These adjustments, however did not affect the functionality provided by VML4Arch. In detail, we had to:

- *Adjust the syntax of some VML4Arch operators.* VML4Arch originally had some operators like `connect c1, c2 using interface i`, which used a concrete syntax slightly different from the standard concrete syntax for VML* operators. We had to adjust the concrete syntax of these operators to fit the standard scheme generated by VML*. For example, the `connect` operator from above now is expressed as `connect (c1, c2, i).`
- *Extend some operator definitions to allow for the use of pointcut expressions as parameters.* VML4Arch originally used direct references to model elements rather than pointcut expressions. This meant that we had to modify some of the operator definitions so that they would be able to deal with receiving sets of model elements as parameters rather than individual model elements only.

5 Related Work

The work presented in this paper is related to work in two areas of research: systematic development of families of languages and support for variability management in SPLE. As the main focus of this paper is on constructing a family of languages, we will begin by discussing literature from this area.

A number of research projects—for example, CAFÉ, Families, or ESAPS—have explored the notion of software system families (or product lines). In this work, we are extending these ideas to families of software languages, specifically for the case of VML languages.

Families of languages have been presented in the research literature for a range of domains: Voelter presents an approach for a family of languages for architecture design at different levels of abstraction [31], Akehurst et al. [32] present a redesign of the Object Constraint Language as a family of languages of different complexity, Visser et al. [33] present WebDSL, a family of interoperating languages for the design of web applications. All approaches, including ours presented in this paper, use very different kinds of technologies for their specific case: Voelter uses conditional compilation to construct an appropriate infrastructure, Akehurst et al. use a special parser technology that enables modular language specification, Visser et al. use rewriting of abstract syntax trees and our approach generates a monolithic infrastructure for each language. Equally, all approaches focus on different purposes of the language family: the different members of the family presented by Voelter are architectural languages at different levels of abstraction. The family presented by Akehurst et al. modularises different features of the OCL language, so that specific languages can be constructed as required for a project. WebDSL is a set of interoperating languages with purposes ranging from data modelling to workflow specification. The family of languages presented in our paper consists of languages that share a common set of core concepts, but adapt these to different languages with which they interface. At this point, an overview of the different potential uses of families of languages begins to emerge. What is needed next, is research into systematic development of such language families beyond individual examples.

Ziadi et al. [10] and Botterweck et al. [11] both propose the implementation of product derivation processes as model transformations. Their proposal relies on the realization of product derivations via a model transformation language. This strategy requires SPL engineers to deal with low-level details of model transformation languages. Our approach provides syntax and abstractions familiar to the SPL engineers. This eliminates the burden of understanding the intricacies associated with model transformation languages and metamodels. A VML* specification is automatically compiled into an implementation of the product derivation process in a model transformation language, but SPL engineers need not be aware of this generation process.

In [12] we have presented a process for developing variability management languages. The structure of these languages has some similarities to the languages developed using VML*, in fact VML4Arch was previously developed based on this process. However, focusing on process rather than infrastructure, [12] falls short of solving the issues discussed in the introduction. In particular, reuse between individual languages is only possible based on a copy-and-paste approach, variability-model and target-model access are closely intertwined with the other infrastructure code

making it difficult to modify them independently. In contrast, in this paper we have presented an infrastructure, which tackles all of these issues. The code generated for a specific VML* language is partially based on code developed for VML4Arch following the process from [12].

Czarnecki and Antkiewicz [6] present an approach similar to ours based on using feature models to model variability. They create a template model, which models all products in the product line. Elements of this model are annotated with so-called presence conditions. Given a specific configuration, each presence condition evaluates to true or false. If a presence condition evaluates to false, its associated model elements are removed from the model. Thus, such a template-based approach is specific to negative variability, which might be critical when a large number of variations affect a single diagram. Our approach can also support positive variability by means of actions such as connect or merge. Moreover, presence conditions imply introducing annotations into the SPL model. Therefore, the actions associated with a feature selection are scattered across the model, which could also lead to scalability problems. In our approach, they are well-encapsulated in a VML* specification, where each variant specifies the actions to be executed. FeatureMapper [7] is another approach similar to that of Czarnecki and Antkiewicz and our approach, avoiding the pollution of the SPL model with variability annotations. FeatureMapper is generic for all EMF-based models and generically integrates into GMF-based editors. In contrast, our approach is based on languages that are specific to a kind of feature model and a kind of target model. Genericity is achieved through a generative approach to creating the infrastructure for these languages from a set of common core concepts. The actual variability model in FeatureMapper is created implicitly by the designer selecting model elements in an editor and associating them with so-called feature expressions determining when the model element should be present in a product model. Negative variability is easily supported by this approach, as model elements can be easily removed if their feature expression is not satisfied by a specific configuration. Positive variability is more difficult to implement: instead of mapping features to target model elements, they need to be mapped to elements of a model transformation, again requiring SPL designers to have sufficiently detailed knowledge of that model-transformation language and the metamodels involved. In contrast, in our approach, designers of a specific VML* language can provide powerful actions that can support both negative and positive variability (or any mixture of the two) in a systematic manner. Finally, Haugen et al. [34] define the common variability language (CVL), which is a generic extension to DSLs for expressing variability. It provides three generic operators, but using these to express variability can lead to comparatively complex models. On the flip side, a VML* language is potentially less flexible than the two other approaches discussed in this paragraph, as it can only support the variability mechanisms for which a corresponding action has been defined.

A completely different approach to SPLE is followed in the feature-oriented software development community. Here, features are directly related to separate modules implementing each feature, where these feature modules can be understood as program or model transformations (e.g., [35]). This implies that no mapping from features to target models is required. Instead, the programming or modelling language must be sufficiently powerful to support modularizing of features as coherent well-encapsulated units of compositions. In another publication [36], we have presented a

feature-oriented approach towards SPL development. In this context, we also noted that a pure feature-oriented approach can lead to a large number of small feature modules negatively impacting scalability and comprehensibility of the approach, especially where features are often associated with small-grain changes to the architecture or implementation. Thus, for such cases, an approach with an explicit mapping may be beneficial.

Generally, all SPL approaches face the problem of ascertaining that only consistent and well-formed product models and implementations can be constructed. This problem becomes even worse when several interconnected types of models representing different views of the system are used—for example, activity diagrams and class diagrams. As a consequence, there is a need to analyse the changes of each view and the inconsistencies that these may cause with other views when instantiating a product model. In our work on VML*, we have not discussed this issue so far, but some previous work on this topic exists from other groups—for example, [37-38].

6 Conclusions

This paper presented a generative approach to building a family of languages for specifying the relationship between variability models and other models in software-product-line engineering. Our experience shows that the proposed infrastructure is powerful enough to support generating different language instances (in addition to the two languages presented here, we are currently developing VML* languages for mapping to openArchitectureWare workflows as well as a number of project-specific DSLs) and that it can reduce the effort required to learn about the support infrastructure for such languages. Specifically, regarding the challenges we identified in Sect. 2.2, our generative approach to the family of VML* languages provides the following solutions: reuse is substantially improved over a copy-and-paste approach as all reusable parts of the infrastructure are encoded in the generator and all variable parts are explicitly configured through language descriptors (Challenge 1). Because all dependencies on varying variability and target models have been made explicit in the language descriptor, model access code could be completely disentangled from the actual model manipulation code (Challenges 2 and 3).

In implementing our prototype, we identified a need for aspect-oriented code generation beyond what is offered by current code-generation engines. Our system is structured such that the code generators for the basic VML* infrastructure and for each evaluation aspect are kept in separate modules. This is sensible because evaluation aspects can be included or excluded from a specific VML* language as required. For some files generated (for example, for plugin descriptors contained in `plugin.xml` files) there is a conflict between code generators for the evaluation aspects: each evaluation aspect needs to contribute to the final contents of the file. Using separate code-generation templates for each evaluation aspect would result in a file containing only the contributions from one evaluation aspect. Aspect-oriented code generation could provide a solution here: it effectively allows the results of two or more different generators to be merged into one output file. However, all current aspect-oriented code generators [18, 39] only support asymmetric aspect orientation.

This requires one template to be declared as the base template while the other templates are aspect templates. These aspect templates can then manipulate generation rules in the base template, providing before, after, and around advice for code generation. For our purposes this is not appropriate; because evaluation aspects may be included or excluded as required, we cannot rely on any one of them being present. Consequently, no template defined for an evaluation aspect can be made into the base template. As the basic VML* generator does not provide a template for `plugin.xml`, this can also not be designated as the base template. For our prototype, this problem has been solved by breaking the encapsulation of evaluation-aspect code generators in a controlled way. However, a cleaner solution using a more symmetric approach to aspect-oriented code generation remains for future work.

References

[1] K. Pohl, *et al.*, *Software Product Line Engineering: Foundations, Principles and Techniques*. Berlin, Germany: Springer, 2005.

[2] P. Clements and L. M. Northrop, *Software Product Lines: Practices and Patterns*. Boston, MA, USA: Addison-Wesley, 2002.

[3] K. Czarnecki and U. W. Eisenecker, *Generative Programming: Methods, Tools, and Applications*: ACM Press/Addison-Wesley Publishing Co., 2000.

[4] K. Kang, *et al.*, "Feature-Oriented Domain Analysis (FODA) Feasibility Study," Software Engineering Institute, Technical report, CMU/SEI-90-TR-0211990.

[5] M. Alférez, *et al.*, "A Model-Driven Approach for Software Product Lines Requirements Engineering," in *proceedings of the 20th International Conference on Software Engineering and Knowledge Engineering*, San Francisco Bay, USA, July 2008, pp. 779-784.

[6] K. Czarnecki and M. Antkiewicz, "Mapping Features to Models: A Template Approach Based on Superimposed Variants," in *Proceedings of the 4th International Conference on Generative Programming and Component Engineering*, Tallinn, Estonia, September-October 2005, pp. 422-437.

[7] F. Heidenreich, *et al.*, "FeatureMapper: mapping features to models," presented at the Companion of the 30th international conference on Software engineering, Leipzig, Germany, 2008.

[8] D. Batory, *et al.*, "The Objects and Arrows of Computational Design," in *Proceedings of the 11th international conference on Model Driven Engineering Languages and Systems*, France, Toulouse, 2008, pp. 1-20.

[9] S. Soares, *et al.*, "Supporting software product lines development: FLiP - product line derivation tool," presented at the Companion to the 23rd ACM SIGPLAN conference on Object-oriented programming systems languages and applications, Nashville, TN, USA, 2008.

[10] T. Ziadi and J. M. Jézéquel, "Software Product Line Engineering with the UML: Deriving Products," *Software Product Lines 2006*, pp. 557-588.

[11] G. Botterweck, *et al.*, "Model-Driven Derivation of Product Architectures," in *Proceedings of the 22nd International Conference on Automated Software Engineering (ASE)*, Atlanta (Georgia, USA), November 2007, pp. 469-472.

[12] P. Sánchez, *et al.*, "Engineering Languages for Specifying Product-Derivation Processes in Software Product Lines," presented at the Software Language Engineering 2008, Toulouse, France, 2008.

[13] N. Loughran, *et al.*, "Language Support for Managing Variability in Architectural Models," in *Proc. of the 7th Int. Symposium on Software Composition (SC)*, March 2008, pp. 36-51.
[14] M. Voelter and I. Groher, "Product Line Implementation using Aspect-Oriented and Model-Driven Software Development," in *Procceedings of the 11th International Software Product Line Conference (SPLC),*, Kyoto (Japan) September 2007, pp. 233-242.
[15] K. Czarnecki and U. W. Eisenecker, *Generative Programming: Methods, Tools, and Applications*: Addison-Wesley, 2000.
[16] P. Jayaraman, *et al.*, "Model Composition in Product Lines and Feature Interaction Detection Using Critical Pair Analysis," in *Proc. of the 10th Int. Conference on Model Driven Engineering Languages and Systems (MoDELS)*, Nashville, (Tennessee, USA) September-October 2007, pp. 151-165.
[17] F. Jouault and I. Kurtev, "Transforming Models with ATL," in *Satellite Events at the MODELS 2005 Conference*, Montego Bay, Jamaica, 2005, pp. 128-138.
[18] *OpenArchitectureWare*. Available: http://www.openarchitectureware.org/
[19] G. Taentzer, "AGG: A Graph Transformation Environment for Modeling and Validation of Software," in *Proceedings of the 2nd Int. Workshop on Applications of Graph Transformations with Industrial Relevance (AGTIVE)*, Charlottesville, (Virginia, USA), September-October 2003, pp. 446-453.
[20] K. Czarnecki, *et al.*, "Staged Configuration Using Feature Models," in *Proceedings of the 3rd International Software Product Line Conference (SPLC 2004)*, Boston, MA, USA, pp. 266-283.
[21] M. Volter and T. Stahl, *Model-Driven Software Development*. Glasgow, UK: Wiley, 2006.
[22] *VML* Download*. Available: http://www.steffen-zschaler.de/publications/vmlstar/
[23] M. Alférez, *et al.*, "A Metamodel for Aspectual Requirements Modelling and Composition," http://ample.holos.pt/gest_cnt_upload/editor/File/public/AMPLE_WP1_D13.pdf, AMPLE D1.3, 2007.
[24] M. Alférez, *et al.*, "Multi-View Composition Language for Software Product Line Requirements," in *Proceedings of the 2nd Int. Conference on Software Language Engineering (SLE)*, Denver, USA, 2009.
[25] A. Sousa. (2008, *AMPLE Traceability Framework Frontend Manual*. Available: http://ample.di.fct.unl.pt/Front-End_Framework/ATF%20Front-end%20Manual.pdf
[26] *Generative Software Development Group, U. Waterloo, Feature Modelling Plugin (FMP) for Eclipse*. Available: http://gsd.uwaterloo.ca/projects/fmp-plugin/
[27] (2009, *VML* Download*.
[28] G. Kotonya and I. Sommerville, *Requirements Engineering: Processes and Techniques*: John Wiley, 1998.
[29] S. Ian and S. Pete, *Requirements Engineering: A Good Practice Guide*: John Wiley and Sons, 1997.
[30] T. Young, "Using AspectJ to Build a Software Product Line for Mobile Devices - www.cs.ubc.ca/grads/resources/thesis/Nov05/Trevor_Young.pdf," University of Waterloo, 2005.
[31] M. Voelter, "A Family of Languages for Architecture Description," presented at the Conference on Object-Oriented Programming, Systems, Languages, Orlando, Florida, 2008.
[32] D. H. Akehurst, *et al.*, "Supporting OCL as part of a Family of Languages," in *Proceedings of the MoDELS'05 Conference Workshop on Tool Support for OCL and Related Formalisms - Needs and Trends*, 2005.

[33] E. Visser, "WebDSL: A Case Study in Domain-Specific Language Engineering," presented at the I*nternational Summer School on Generative and Transformational Techniques in Software Engineering (GTTSE 2007)*, Heidelberg, October 2008.

[34] Ø. Haugen, *et al.*, "Adding Standardized Variability to Domain Specific Languages," in *Proceedings of the Conference on Software Product Lines (SPLC'08)*, 2008, pp. 139-148.

[35] D. Batory, *et al.*, "Scaling Step-Wise Refinement," in *IEEE Transactions on Software Engineering*, 2003, pp. 355-371.

[36] L. Fuentes, *et al.*, "Feature-Oriented Model-Driven Software Product Lines: The TENTE approach," in *Proceedings of the Forum of the 21st International Conference on Advanced Information Systems (CAiSE)*, Amsterdam, The Netherlands, 2009.

[37] S. Thaker, *et al.*, "Safe Composition of Product Lines," in *Proceedings of the 6th International Conference on Generative Programming and Component Engineering (GPCE)*, Salzburg, Austria, 2007, pp. 95-104.

[38] M. Janota and G. Botterweck, "Formal Approach to Integrating Feature and Architecture Models," in *Fundamental Approaches to Software Engineering (FASE)*, Budapest, Hungary, 2008, pp. 31-45.

[39] *MOFScript*. Available: http://www.eclipse.org/gmt/mofscript/

11

Model-Driven Requirements Specification for Software Product Lines

Authors: Mauricio Alférez, Ana Moreira, Vasco Amaral, João Araújo.

Paper Summary: This book chapter provides an overview of different approaches for specifying requirements models and composing models for specific products of an SPL. In particular, it emphasizes one of the most recurring specification techniques: model-driven and use case scenario-based specification. This technique, in combination with feature models and the Variability Modeling Language for Requirements (VML4RE), integrates our approach for model-driven requirements specification for SPLs. This book chapter motivates and describes some of the patterns followed by our approach as well as it gives an easy to understand usage overview.

Authors Contribution: Mauricio Alférez was the main author and responsible for the main part of the research and writing of this paper, accounting for the 90% of the work. Other authors gave interesting comments that helped to improve the content of the paper.

Publication Arena: Published in the book "Model-Driven Domain Analysis and Software Development: Architectures and Functions, 2011" [13]

Model-Driven Requirements Specification for Software Product Lines

Mauricio Alférez, Ana Moreira, Vasco Amaral, João Araújo

Centro de Informática e Tecnologias da Informação, Departamento de Informática, Faculdade de Ciências e Tecnologia, Universidade Nova de Lisboa, Portugal

***Abstract*—Model-driven methods for requirements specification in Software Product Lines (SPLs) support the construction of different models to provide a better understanding of each SPL feature and intended use scenarios. However, the different models must be composed to show the requirements of the target applications and, therefore, help to understand how features will be integrated in a new product of a software product line. Although well-established standards for creating metamodels and model transformations exist, there is currently no established foundation that allows practitioners to distinguish between the different modeling and composition approaches for requirements models. This chapter provides an overview of different approaches for specifying requirements models and composing models for specific products of an SPL. In particular, it emphasizes one of the most recurring specification techniques: model-driven and use case scenario-based specification. This technique, in combination with feature models and the Variability Modeling Language for Requirements (VML4RE), integrates our approach for model-driven requirements specification for SPLs.**

***Index Terms*—Variability management, requirements engineering for software product lines, model-driven requirements specification, scenarios specification, UML, VML4RE.**

I. Introduction.

Software Product Lines are increasingly being adopted by major and medium-sized industrial players to quickly address change requests and improving time to market. SPLs enable modular, coarse-grained reuse through a set of core and varying software elements addressing a particular application domain (Clements & Northrop, 2002). Software Product Line (SPL) engineering is a promising approach to increase software quality and productivity. It encompasses the creation and management of families of products for a particular domain, where each product in the family is derived from a shared set of core assets, following a set of prescribed rules (Clements & Northrop, 2002).

An SPL product shares, with other systems, properties or functionalities that are relevant to some stakeholders. These are usually called *features* and express not only commonalities, but also variabilities that allow us to distinguish among products. The term *commonalities* refers to features that are mandatory to every product in an SPL. It is used to reference the parts of the requirements that are related to SPL common features. The term *variabilities* refers to the variable (optional, variation points and variants) features of an SPL. *Optional* features are not mandatory and might not be included in some of the products of an SPL. A *variation point* identifies a particular concept within the SPL requirements specification as being variable and it offers a number of *variants*. A *variant* describes a particular variability decision, such as a specific choice among *alternative variants*. Typically, we can model the available features and their dependencies (e.g., if feature *X* is selected, feature *Y* also must be selected) using a feature model (Czarnecki & Eisenecker, 2000; Kang, Cohen, Hess, Novak, & Peterson, 1990), that helps to capture the commonalities and variabilities of a family of products.

To understand each SPL feature and intended use scenarios of target products, Model-driven methods for requirements specification support the construction of different models that design the product behavior. However, to show the requirements of the target products, different models must be composed to help to understand and communicate to users, managers, testers and programmers the intended behavior of the new product to be produced from the SPL. Although well-established standards for creating metamodels and model transformations such as Meta-Object Facility (OMG, 2009a) exist, there is currently no established foundation for specifying requirements models for SPLs and compositing these models for specific products. This chapter introduces a classification of several existing approaches for model-driven requirements specification for Software Product Lines and focuses on exploiting use scenario-based techniques. We make explicit the different ways of specification taking into account the concrete syntax of the requirements models and the separation of three core components needed to specify and compose requirements models:

- the base models that specify requirements. For example, specifications of use scenarios using use cases models complemented with activity diagrams;
- variability information that makes explicit which are the SPL features that are common to all the products and which are the features that are particular to some products of the SPL; and
- configuration knowledge, which establishes the mapping between features and base models that specify requirements, for example, associating feature expressions, in the form of logical propositions, to specific model fragments. Also, in Model-Driven Development (MDD), configuration knowledge may include the specification of the transformations of SPL requirements models to compose models for specific products.

This chapter is structured as follows. Section "Requirements Modeling for SPLs" provides an overview of both textual and

graphical notations for modeling SPL requirements and different ways to compose requirements. That section motivates this work, showing some shortcomings in previous approaches. Section "A Model-Driven Requirements Specification Approach for SPLs" constitutes the major contribution of this work. It presents our approach for model-driven requirements specification for SPLs. This approach employs use case models and activity diagrams to represent one of the most recurring techniques for requirements modeling such as use case scenario specifications. We employ feature models to model variability information in combination with specially tailored composition rules for requirements models provided by the Variability Modeling Language for Requirements (VML4RE). The final sections of this chapter synthesize our contributions to the field of model-driven requirements specifications for SPLs, address future research directions, and conclude this chapter.

II. Requirements Modelling for SPLs.

The success of a SPL depends on abstraction and decomposition mechanisms supporting modular treatment of its commonalities and variabilities (Alves, et al., 2006; Figueiredo, et al., 2008). SPLs can most easily accommodate changes and be instantiated to specific products if all varying and core software elements are defined in a modular fashion, from requirements to architecture and implementation. Mainstream techniques to support modular realization of software variabilities are focused on code. Typical examples of these techniques are object-oriented mechanisms, design patterns, and conditional compilation (Alves, et al., 2006; Figueiredo, et al., 2008). However, less attention has been given to the use of models as a key asset during requirements engineering.

One of the most recurring techniques used to specify requirements both in single and SPL systems is use case scenario modeling (Alexander & Maiden, 2004; Cockburn, 2001; Jacobson, 1992). Each use case describes how actors (i.e., persons, organizations or other (sub)systems) will interact with the system to be developed to achieve a specific goal. One or more scenarios may be generated from a use case, corresponding to the detail of each possible way to achieve that goal. Scenarios can help to (Alexander & Maiden, 2004; Cockburn, 2001):

- validate requirements;
- generate acceptance criteria for requirements;
- verify the level of abstraction of each requirement;
- help to create initial system architectures.

A number of different approaches for use case scenario variability modeling have been proposed for SPLs, such as (Alférez, Santos, et al., 2009; Bonifacio & Borba, 2009; Czarnecki & Antkiewicz, 2005; Eriksson, Börstler, & Borg, 2005; Gomaa, 2004). These approaches differ in their notation which is based on graph models such as use case and activity models (OMG, 2009b), or text, such as the black-box notation to represent use case scenarios (Alexander & Maiden, 2004). Each of the approaches has its own advantages and drawbacks. A free-format textual representation of requirements usually requires a short-time of elaboration at the expense of some ambiguity due to interpretation of the natural language. On the other hand, using a graph-based representation allows using a standard language to express the requirements, such as, conditionals, control flows and parallelism, therefore, contributing to avoid ambiguity.

In addition to choosing between textual or graphical notations for the requirements specifications, developers have to decide whether or not to mix requirements specifications and variability data in their SPL models. Adding variability data to textual or graph-based requirements models is used to make explicit the relationships between their fragments and SPL features. Next, we will briefly introduce both approaches using both text- and graph-based requirements models. In these approaches we start with free format textual-based requirements descriptions. This approach was the base for more structured approaches such as use scenario-based approaches that can be supported by Model Driven Development (MDD) techniques. Our objective with this is to establish a foundation that allows to distinguish between the different modeling approaches for requirements models and also to motivate the use of model-driven techniques.

A. Mixing Variability into the Requirements Specification.

1) Unformatted Textual Specifications.: Natural language can be used to express optional and alternative parts in the specifications using phrases related to SPL features. For example, the sentence: "*the Security System should support either PIN entry via Keypad, Fingerprint Scanning or Retina Scanning*", does not make clear the number of alternatives that different clients may require for their products. For example, some may select the cheaper alternative for his product; others may prefer a more sophisticated one integrated in their solution, while others may want any possible combination of the three for extra security. After successive changes in the specification, the number of possible choices and their impact in other features may become complex to control, and the use of natural language will result inefficient and insufficient to express correctly and unambiguously the variable parts in the SPL. Therefore, even when the requirements are carefully written to avoid ambiguities, it is virtually impossible in a large project to assure their quality after successive modifications.

2) Structured Textual Specifications.: Given what was discussed in the last section, a way to document variability in a more concise and unambiguous manner is necessary. Table I (a) exemplifies a more understandable way to express the requirements based on formats similar to those provided by (Gomaa, 2004; Pohl, 2006). This structured textual format also shows which and how many alternatives of a variation point can be selected for a specific product.

Therefore, structuring the requirements specifications by expressing the kind of variability with key phrases such as "only one", "two from all", etc., reduces the ambiguity of a simply unformatted textual requirements approach. This intuitive approach can be complemented with a textual language to describe variability and features. Using a structured language usually allows reducing the size of the specification and helps to restrict the way in which the developer elaborates

(a)

SPL-Requirement # *X*. The security system shall support only one of the following alternative devices for user authentication:

V1 (Variation point #1): Authentication Device
Only one of the following variants can be selected for **V1**:

V1.1 PIN using Keypad
V1.2 Fingerprint Scanner
V1.3 Retina Scanner

(b)

SmartHome : **all** (electronicWindows, security?)
Security : **all** (AuthenticationDevice, AlarmType)
AuthenticationDevice : **one-of** (keypad, fingerprintScanner, retinaScanner)
AlarmType : **more-of** (siren, light)

Table I
(A) EXPRESSING VARIABILITY TEXTUALLY; (B) EXPRESSING VARIABILITY TEXTUALLY USING A DESCRIPTION LANGUAGE.

the requirements specification. Consequently, in the case of textual requirements descriptions, each model written using a structured language will have to follow some construction rules. These rules are defined by the language grammar that performs the role of a *metamodel*, as it is known in the MDD domain.

Table I (b) shows a short example, inspired in the Feature Description Language (FDL) proposed by Deursen and Klint (Deursen & Klint, 2001). The base of FDL consists of a number of feature definitions which is the feature name followed by ":" and the feature expression. In this example the variation points start with upper case while the variants (e.g., keypad, fingerprints, retinaScanner) and optional features (e.g., security) start with lower case in the feature expression side. Also, supposing that security is an optional feature, its name is followed by the "?" sign. However, while this approach works well for small problems, it can be difficult to keep an overview of variabilities and dependencies as the number of features and options increases. Furthermore, many times it is necessary to express by means of additional models, usually based on graphs, what is the expected behavior of the features. Also, it is necessary to specify how the presence or absence of specific sets of features affects the behavior of the target system.

3) Graph-Based Specifications.: The discussion above highlights the need for a more efficient solution. For this reason, graph-based models are sometimes preferred to design and communicate the requirements of the system. A number of authors such as Gomaa (Gomaa, 2004), and Bragança and Machado (Bragança & Machado, 2007) add variability information into the requirements models. Currently, there is no standard that restricts the way in which the variability information should be added and represented in the concrete syntax of the requirements models. However, the use of stereotypes and model annotations is a common practice for both UML-based models and a number of Domain Specific Languages (DSL) (DSM forum website). Stereotypes and annotations help to make explicit what parts of the models are common to all the products of the product line (also called *kernel*, *mandatory* parts, or *commonalities*) and which parts are variable (i.e., not common) among all the products (also called *variabilities*).

Figure II.1 is based on the notation proposed by Gomaa (Gomaa, 2004) and uses stereotypes such as "Kernel" and "Alternative" to mark what is common and what the alternatives are. Similarly, supposing that not all the products will include the security feature, the security package is labeled with an "Optional" stereotype.

This kind of diagrams could be complemented with cardinality using, for example, UML notes as shown by (Bragança & Machado, 2007). For example, a note showing a cardinality of "1..3" attached to the "Authenticate User" would express that a minimum of 1 and a maximum of 3 alternatives could be chosen to be included in a single product. This kind of specifications has some drawbacks. First, the models could be polluted with stereotypes, some of them to indicate SPL variability and others to express standard UML semantics, such as "extends" and "includes". Hence, it would not be clear which "extension" use cases are created to offer possible alternative use case scenarios for a single product, and which use cases are created to show SPL alternatives, for example. This situation may become complex if the modeller is using a UML profile to model in a specific domain that requires the use of additional stereotypes. Therefore, to augment the models' understandability, it would be better to follow a separation

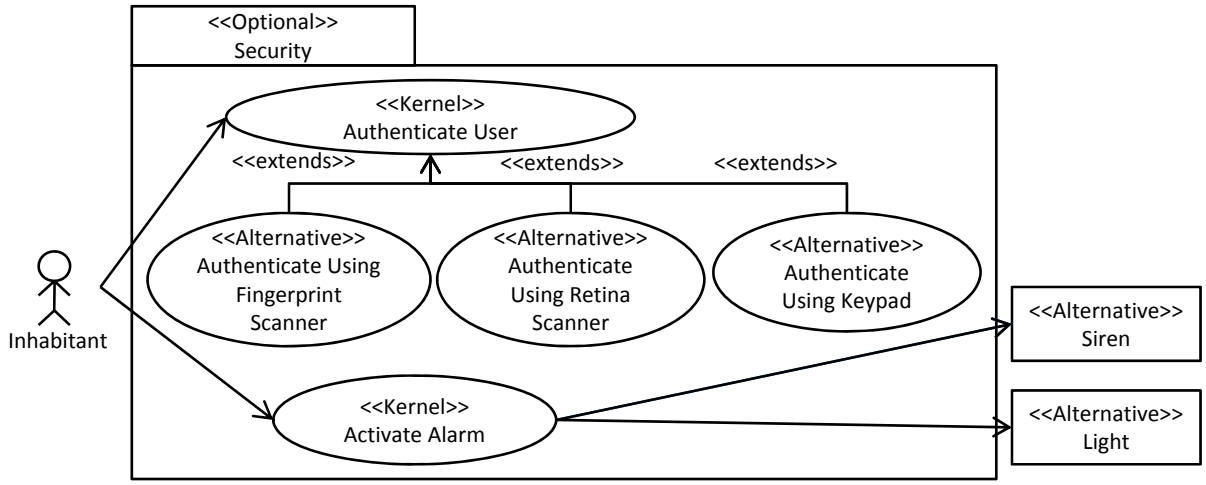


Figure II.1. Expressing variability graphically using stereotypes.

of concerns approach (Filman, Elrad, Clarke, & Aksit, 2004; Moreira, Rashid, & Araújo, 2005) and separate variability information from the concern requirements specification.

B. Separating Variability Information and Requirements Specification.

There are some patterns that have been followed when variability information and requirements specifications are separated. The first one is about linking parts of the requirements specification (also called model fragments) to a feature model. Feature models (Czarnecki & Eisenecker, 2000; Kang, et al., 1990) are a well accepted means for expressing requirements in a domain on an abstract level. They are applied to describe variable and common features of products in a product line, and to derive and validate configurations of software systems. The second pattern is about the use of mechanisms of composition based on adding or removing optional model fragments according to features' selections in a feature model.

1) Linking Requirements Models to Features Models.: One of the patterns used as prerequisite to compose models for specific products is to link fragments of the requirements models to features in a feature model. These links enable the adaptation of base requirements models based on features' selection. A feature model groups hierarchically from general to more specific common and variable features using a refinement relationship. This may also include cardinality as in (Czarnecki, Helsen, & Eisenecker, 2004) to express properties such as the number of alternative features that can be chosen for specific variation points, that is for example, "the number of authentication devices that can be included to guarantee a secure system". On the other side, requirements models, such as use cases and activity diagrams, express the intended behavior of a system focusing on the clear representation of use scenarios to the SPL and products' stakeholders.

The base mechanism to link requirements model fragments to features is to use a correspondence table (also called: mapping table), as presented by (Gomaa, 2004), (Pohl, Böckle, & van der Linden, 2005) and (Alférez, et al., 2008). Also, with the advent of MDD technologies, the use of tools such as FeatureMapper (Heidenreich, 2010), FMP (Antkiewicz & Czarnecki, 2004; Czarnecki & Antkiewicz, 2005), and VML4RE (Alférez, Santos, et al., 2009) ease the linking between features and other models. The mechanisms used for ease the linking between requirements models and features range from more usable visual editors with facilities such as drag-and-drop and multiple selection of model fragments, to mechanisms to create links programmatically, for example, using quantification and queries on the models' properties to determine correspondence between features and model fragments. The goal of all these mechanisms is to ease the arduous task of creating and maintaining links manually between many features and model fragments as happens with the use of simple mapping tables.

2) Composition of Requirements Models.: After linking fragments of the requirements models to features in a feature model, the model fragments can be composed to show the requirements of the target applications and, therefore, help to understand how features will be integrated in a new product of a software product line. The way of composing the target model of a product is based on removing (i.e., *Negative Variability*) or adding (i.e., *Positive Variability*) model fragments to a base requirements model. Next we explain these mechanisms.

a) Negative Variability Composition Mechanism.: Negative variability selectively takes away parts of a model based on the presence or absence of features in configuration models, such as feature model configurations (Volter & Stahl, 2006). When using negative variability, developers have to model the overall SPL requirements. Then, after linking model fragments to certain features in a feature model, some model fragments are taken away from the base model according to a certain features' selection for a particular product. Typically, requirements composition approaches have followed negative variability mechanisms. Some examples of approaches using mostly negative variability mechanisms are proposed by Eriksson (Eriksson, et al., 2005) and Czarnecki and Antkiewicz (Czarnecki & Antkiewicz, 2005).

b) Positive Variability Composition Mechanism.: To reduce requirements models composition to a simple removal of some fragments (e.g., a use case, a scenario step, etc.) according to a specific feature model selection (also called feature model configuration) is straightforward. However, it may difficult to specify the overall SPL requirements at the beginning of the development process. The composition mechanism where the target model is composed adding model fragments to a base model is called "positive variability". This starts with a minimal base model and selectively adds additional parts (Volter & Stahl, 2006). The base model generally represents the model fragments that are common to all products within the product line. Model fragments related to varying features are attached to the requirements models based on the presence or absence of features in the configuration models, such as feature model configurations. Some authors that propose approaches that employ positive variability mechanisms are Eriksson (Eriksson, 2006) that instantiate parametric features with specific values in use case scenario descriptions, and Alférez et. al. (Alférez, Santos, et al., 2009) that add and modify specific model fragments in use cases models and activity diagrams.

However, to compose requirements models for a specific product may require to insert and remove model fragments. One situation when it may happen is when we need to insert model fragments related to a feature that is incompatible with other feature that was already added into the base model. Therefore, composition requires to remove and to insert model fragments. This demands the need of richer composition mechanisms that allow both negative and positive variability to derive target requirements models as we show later in this chapter.

c) Other Mechanisms to Support Variability Composition.: Apart from the use of positive and negative variability mechanisms, there are situations in which the adaptation of the fragments in a model depends on the simultaneous presence and absence of a specific combination of features. Czarnecki and Antkiewicz (Czarnecki & Antkiewicz, 2005) recognize this fact. They expressed use case scenarios using UML activity diagrams and annotate them with presence conditions. These presence conditions are expressions that can be evaluated to false or true according to the set of features selected for a specific product. Although this approach uses a separate feature model to express variability information, this, as well as Eriksson's approach (Eriksson, 2006), fails to fully separate "configuration knowledge" from the requirements specifications. The result is that the requirements models turn out full of textual annotations expressing presence conditions for each fragment of the models, including fine-grained elements like "actions" and control flows.

III. A Model-Driven Requirements Specification Approach for SPLs.

Until now we have mentioned three elements in the definition of Model-Driven Requirements Specification for Software Product Lines:

- *Requirements specification* that defines use scenarios describing the expected behavior of the SPL's members. These scenarios might be optional, have parameters, and modify the behavior of other use scenarios.
- *Feature models* that make explicit what features are common or variable and model the dependencies between features. Feature models also contribute to the composition process, since they are used for checking if a product configuration represents a valid member of the product line.
- *Configuration knowledge* defined as a set of instructions that relates feature expressions to transformations (Czarnecki & Eisenecker, 2000) can be used for automatically generate product models. However, configuration knowledge is sometimes tangled with the base requirements specification raising understandability and maintainability problems.

Next, we describe our approach for keeping separated requirements elements, feature models and configuration knowledge to contribute to the understandability and maintainability of the three types of models. This approach uses models and specially-tailored composition rules to customize use case scenarios based on activity diagrams and use case models. Graph-based models such as use case and activity diagrams help to avoid ambiguity and add more rigor to the specifications (Pohl, et al., 2005).

A. Separation of Concerns in the Definition of Model-Driven Requirements Specification.

The idea of linking features and requirements models' fragments, separating configuration knowledge, and supporting positive and negative variability mechanisms, is used in the Variability Modeling Language for Requirements (VML4RE) (Alférez, Santos, et al., 2009). Figure III.1 sketches the idea behind the separation of concerns marked with the labels 1-4:

1) *SPL requirements specifications* are expressed using different models such as use case models and activity diagrams that express use scenarios. These models conform to the UML metamodel (OMG, 2009b).
2) *Feature model* that makes explicit common and variable features as well as the dependencies between them. Each feature can be associated to fragments in the SPL requirements specification, and particular fragments in the SPL requirements specifications such as use cases, actors, activities, etc, can be associated to several features in the feature model.
3) *Product configuration* contains the selection of features that will be included in the specific product and that will be derived from the SPL.
4) *Configuration knowledge* establishes the mapping between features and base models that specify requirements, and expresses how to compose the base models according to some transformation rules to produce requirements models for target products. This information, together with the dependencies between features described in the feature model (e.g., feature "retina Scanner" excludes feature "fingerprint Scanner"), helps to derive target products correctly from a set of reusable

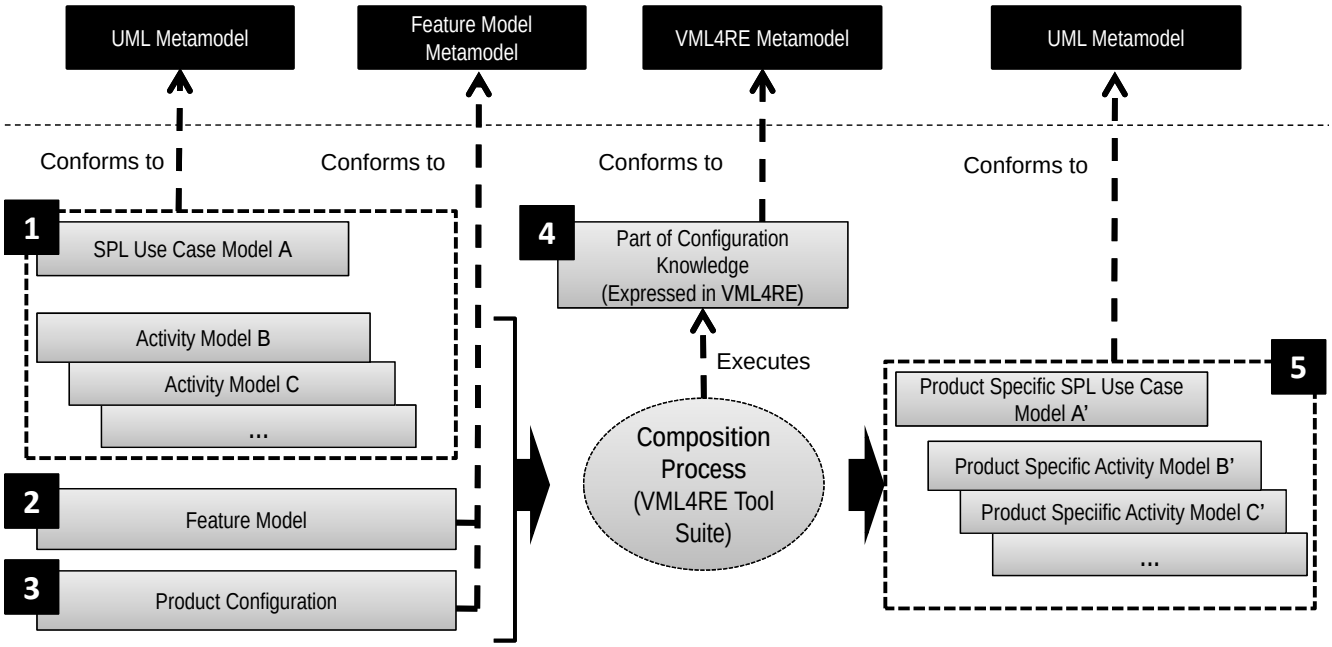


Figure III.1. Linking features to requirements model fragments separating configuration knowledge.

SPL requirements models. Other information that is part of configuration knowledge, and that is usually attached directly to the feature model are the potential bad feature interactions and default values for features selections.

In our approach the metamodel of the source models such as A, B and C in Figure III.1 will be the same for the target models A', B', C', respectively. This means, for example, that if the model B (Figure III.1-1) is an activity diagram representing an SPL use case scenario, the resultant product specific requirements model B' (Figure III.1-5), where B' means a composed model created based on B, will be an activity diagram as well. Product specific models like A', B' or C' are generated for a specific product described by a product configuration (Figure III.1-3). Each product configuration contains a selection of features in the feature model (Figure III.1-2). The composition process is guided by the configuration knowledge. The VML4RE represent part of the configuration knowledge for requirements models that is related with the transformation of the models. This specifies how to link model fragments with features in the feature model programmatically using quantification, and also allows describing a composition workflow. In particular VML4RE uses composition actions specially tailored for graph-based use case scenarios like use case models and activity diagrams.

B. Raising the Abstraction Level to Specify Requirements Models Transformations.

Figure III.2 sketches the idea of using a domain specific language such as VML4RE to raise the level of abstraction to express the composition of the requirements models. In comparison with the technical background required by developers, requirements modelers simply have to employ the familiar vocabulary provided by VML4RE. Naturally, composition actions used by the requirements modeler in VML4RE are mapped to low level transformations of the requirements models. Therefore, to create more composition actions it is necessary the intervention of either a model-driven developer that knows the details of the UML use case and activity diagrams and masters general purpose model transformation languages such as ATL (Jouault & Kurtev, 2005), AGG (Taentzer, 2003), Xpand (openArchitectureWare.org), or that understands how to modify programmatically the base specifications using the Application Programming Interface (API) provided by the respective tool used to model the requirements, e.g., UML tools plug-in for the Eclipse Framework (Eclipse Foundation).

Table II shows examples of a subset of composition actions in VML4RE that are available for the requirements modeler. The names of specific elements in the models are in italics while the keywords of the language are in bold. It is important to note that the composition actions is address positive and negative variability as it was described in the Section "Composition of Requirements Models".

C. Semantics of VML4RE Composition Actions.

The semantics of each VML4RE composition action can be defined in terms of a model-to-model transformation where the metamodel of source and target models is UML. Transformations can be presented by the left hand side (LHS) and right hand side (RHS) graphs. In general, a graph transformation is a graph rule r: $L \rightarrow R$ for LHS graph L to a RHS graph R. The process of applying r to a graph G involves finding a graph monomorphism, h, from L to G and replacing $h(L)$ in G with $h(R)$ (Grzegorz, 1997).

Let's take as an example from (Alférez, Santos, et al., 2009). It shows the "Insert Use Case Links" action using the use case link type "associatedWith", which connects

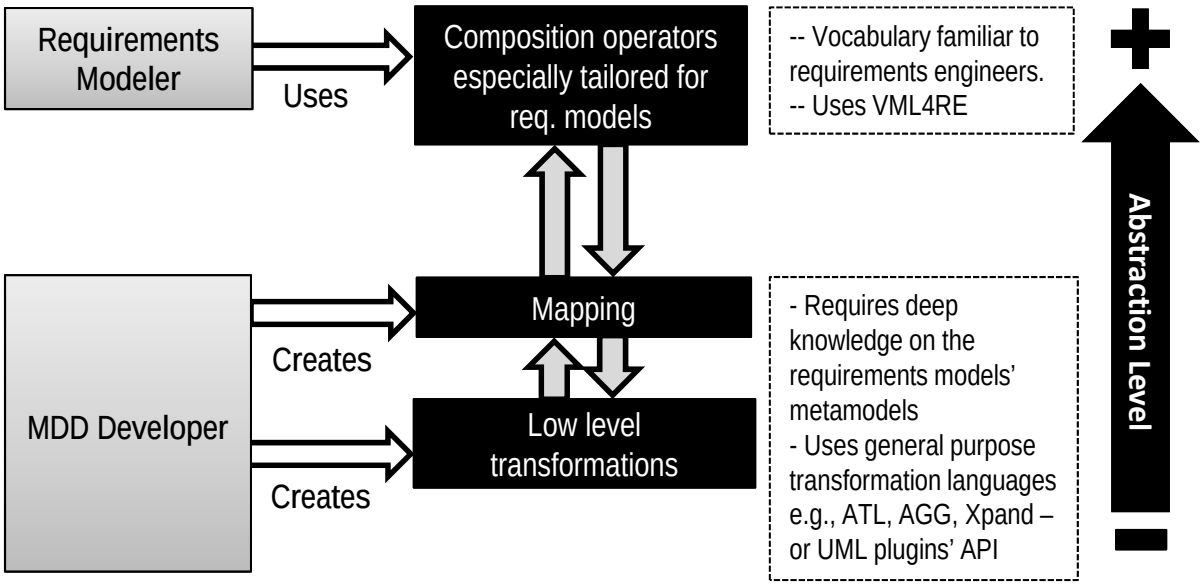


Figure III.2. Raising the abstraction level to ease the use MDD transformations in requirements models.

Insert package, use case, actor, use case links, activity, activities and relationships between activities.

Examples:

```
insert (package packageX into ucModel useCaseModelY);
insert (UClinks_of_type: associatedWith { from actor_X to useCaseA,
        useCaseB, UseCaseC } );
insert (InsertActivityLinks activityA to activityB with guard guard);
insert (InsertActivityLinks activityA to activityC);
insert (actor actorX into ucModel useCaseModelY);
```

Remove the same elements than can be inserted using the **Insert** types.

Replace activities by activities and activity models, actors by actors, use case by use case, etc.

Table II
SOME COMPOSITION RULES IN VML4RE.

an actor and a use case using an association link (insert (UCLinks_of_type: associatedWith { from actorD to useCaseModelA.PackageB.useCaseC});). The intended transformation of the use case model can be presented by the left hand side (LHS) and right hand side (RHS) graphs in Figure III.3, where the inputs are a use case model, a use case, a use case's package, and an actor. If there is already an association between the actor and the use case in the same package, the transformation is not applied to avoid duplicates. This is expressed with the cross in some elements in the LHS graph that act as negative application conditions (NAC). It means that any match against the LHS graph cannot have a *packageB* with any existing association between *actorD* and the *useCaseC*.

The notation used to define this graph transformation is similar to the one used by (Markovic & Baar, 2005) where the LHS and RHS patterns are denoted by a generalized form of object diagrams. However, for visual simplicity we added dashed lines between elements to represent any number of containments (in this case, package's containments). We defer to (Markovic & Baar, 2005) for the readers interested in details of this notation.

It is important to note that VML4RE requirements modelers do not need to build graph-based transformations, know the details of the metamodels, or to master general purpose transformation language or technologies. VML4RE provides requirements-specific composition actions that facilitate the specification of the composition of the base models.

D. Home Automation Case Study.

To clarify the idea and the use of these composition rules, we have modeled a subset of the Smart Home. This is a home automation software product line being developed by

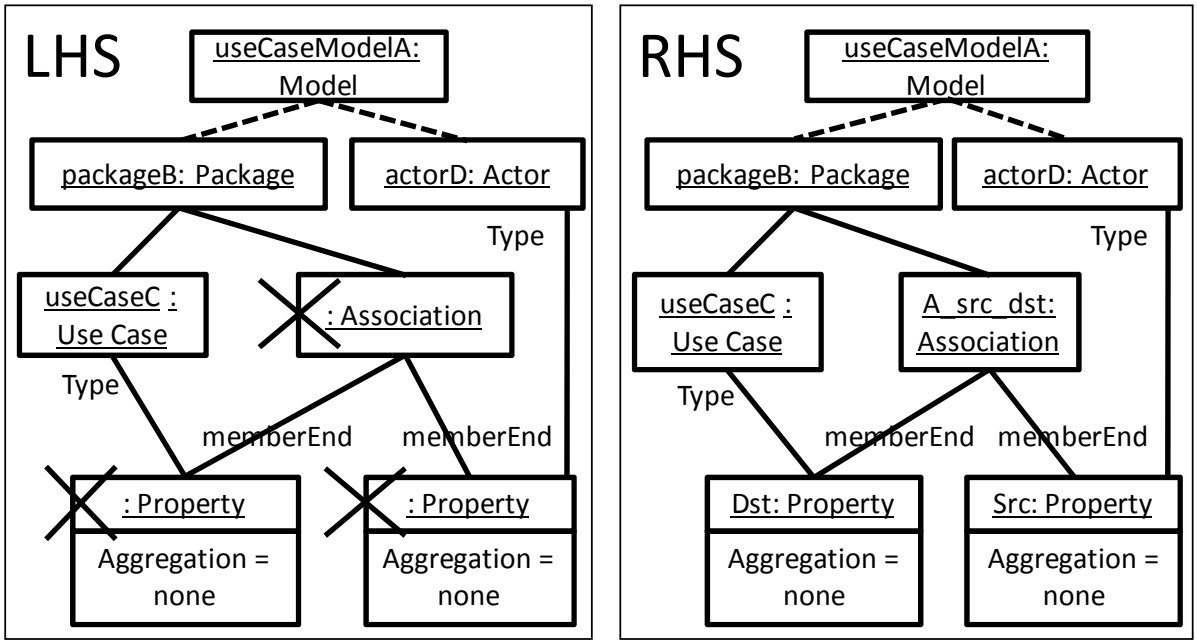


Figure III.3. Graph rule to insert an association between *actorD* and *useCaseC* in *packageB*.

Siemens AG (Morganho, et al., 2008). We already used part of this case study to exemplify the different textual approaches in the Section "Requirements modeling for SPLs" using several authentication devices, keypad, retina scanner, fingerprint scanner. Figure III.4-1 provides a feature model that includes these features as well as others related with security and windows management, while the Figure III.4-2 presents one of its possible configurations, the "Economical Smart Home". Some optional features that are expensive and add more cost to the system are not included in such Economical product. Therefore, retina and fingerprint scanners, as well as cameras, are not part of the final product.

As far as the Security feature is concerned, inhabitants can initiate the secure mode by activating the glass break sensors or/and camera surveillance devices (Glass Break Sensors and Cameras features). If an alarm signal is sent by any of these devices, and according to the security configuration of the house, the Smart Home decides to (i) Secure the house by activating the alarms (Siren and Lights features), and/or (ii) closing windows and doors (Electronic Windows feature) (see use case model in Figure III.4-3). Smart Home can also choose to open or close windows automatically to regulate the temperature inside the house as an option to save energy (Electronic Windows feature). Alternatively to the electronic windows, the inhabitants could always be able to open and close the windows manually (Manual Windows feature). In Figure III.4-4 we show the intended target use case model for the Economic home.

We chose use cases whose detailed behavior is modeled using activity models. This alternative provides models that conform to a metamodel (i.e., the metamodel of UML activity diagrams), thereby reducing the ambiguity in the specifications (Pohl, et al., 2005). The detailed specification of use cases as activity models also enables customizations of use cases realizing specific SPL configurations.

Figure III.5-1 shows a simplified SPL scenario for the "activate secure mode" use case and how it was customized for the Economic home that only have glassbreak sensors as available surveillance device (Figure III.5-2). This figure also shows the links between features in the feature model and specific model fragments, and links between model fragments in different levels of abstraction of the requirements specification that helps to understand the relationships between the different models. The composition of the elements is expressed using a VML4RE model as the one shown in Table III.

The VML4RE specification in Table III expresses how to configure the models for a specific product. Therefore, it contributes to express configuration knowledge as it was sketched in Figure III.1-4. The VML4RE specification references the requirements models and specifies composition rules. In the specification it is possible to use expressions such as "and (glassbreak, not (cameras))" (see Table III - line 8), which means that the composition rules inside brackets will be executed if the expression evaluates to TRUE. This is, when the "glassbreak" sensors feature is included but not the "cameras" feature in the feature model configuration. The VML4RE tool (AMPLE, 2009) receives as input the SPL use cases and activity models, the feature model configuration and the VML4RE specification. As outputs, the VML4RE tool generates: (i) use cases of a product; and (ii) activity models that describe product's use scenarios.

There are composition rules that add elements to the SPL model and there are others that remove or replace elements. The specification in Table III shows familiar vocabulary such as useCase, activity, actor, etc.. The modeler does not require to know metamodel complexities such as the link between the

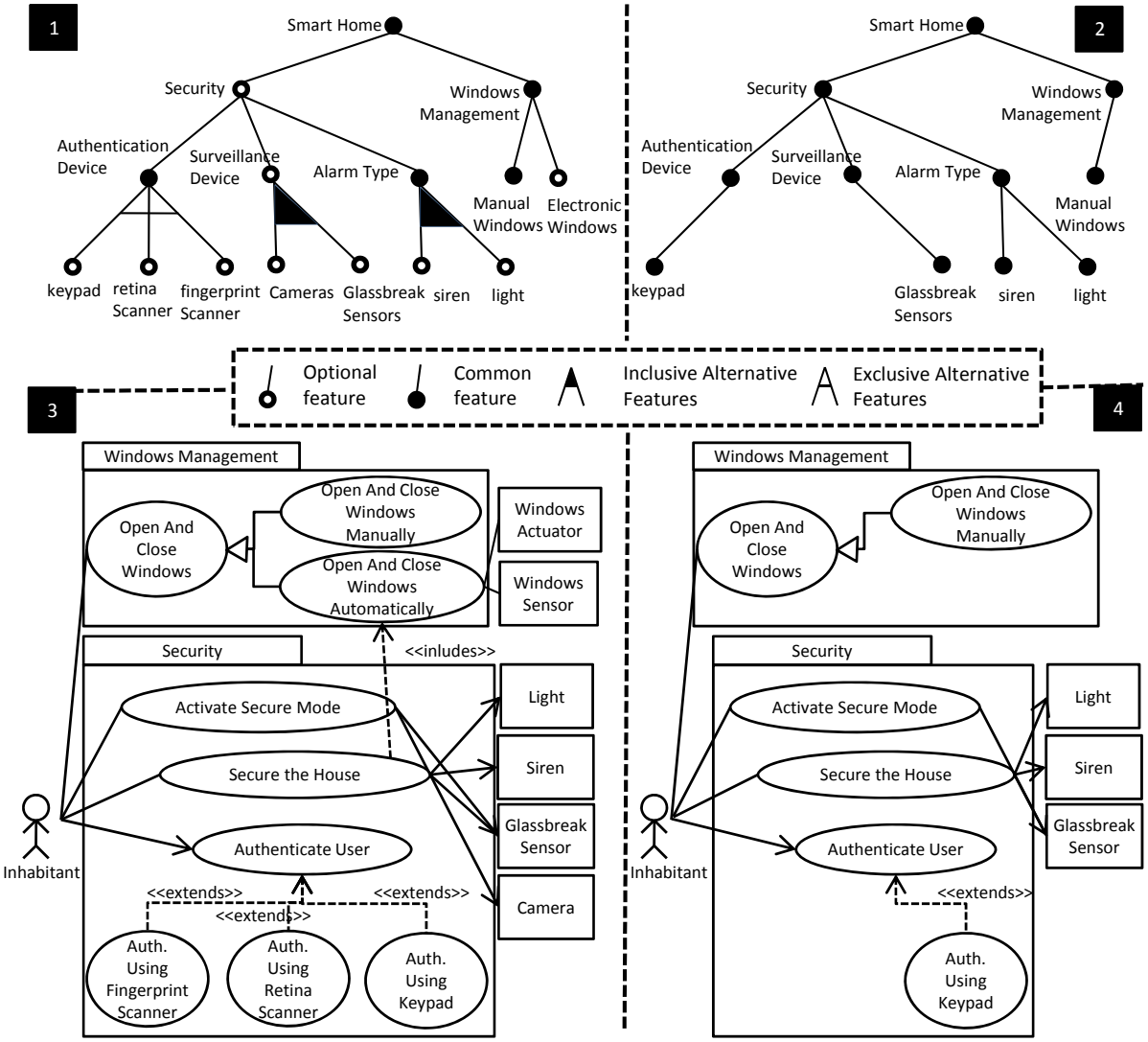


Figure III.4. (1) Smart home feature model; (2) feature model configuration for the economic home; (3) use case model for the Smart Home SPL; (4) use case model for the economical edition of the Smart Home SPL.

UML metaclasses "Use case", "Association" and "Property" to express "simply" things such as for example the insertion of an association relationship between an actor and a use case. Also, the way of expressing what happens when a variant feature is selected or not is very intuitive as it can be appreciated for the case of the variant "cameras" (Table III, lines 25-30).

IV. Contributions of Model-Driven Development to Requirements Engineeering in SPLs

The use of models, metamodels and models transformations in MDD provide the possibility of using models as key assets in the development process. Some of the contributions of using approaches for model-driven development requirements engineering in SPLs such as the one presented in Section "A Model-Driven Requirements Specification Approach for SPLs" are:

- *Rigor of the specifications through the verification of the conformity of the models with their metamodels*: The models that express the requirements specifications conform to metamodels that can be verified. The verification of the models is important as it is required to produce models written in a known metamodel. If the metamodel of the model is known, it is possible to create transformation rules that are described based on the metaclasses of the source and target models' metamodels. Therefore, a chain of transformation rules and additional increments of information in the requirements specification models could be applied to refine them to more detailed models such as design models, or initial architectural models.
- *Employ a correct abstraction level to express the requirements*: the use of domain specific languages such as feature models and VML4RE helps to express efficiently composition, variability and configuration knowledge of requirements models in the vocabulary and in the way that the users understand. This is a big contribution to the solution of the problem of abstraction mismatch (Sánchez,

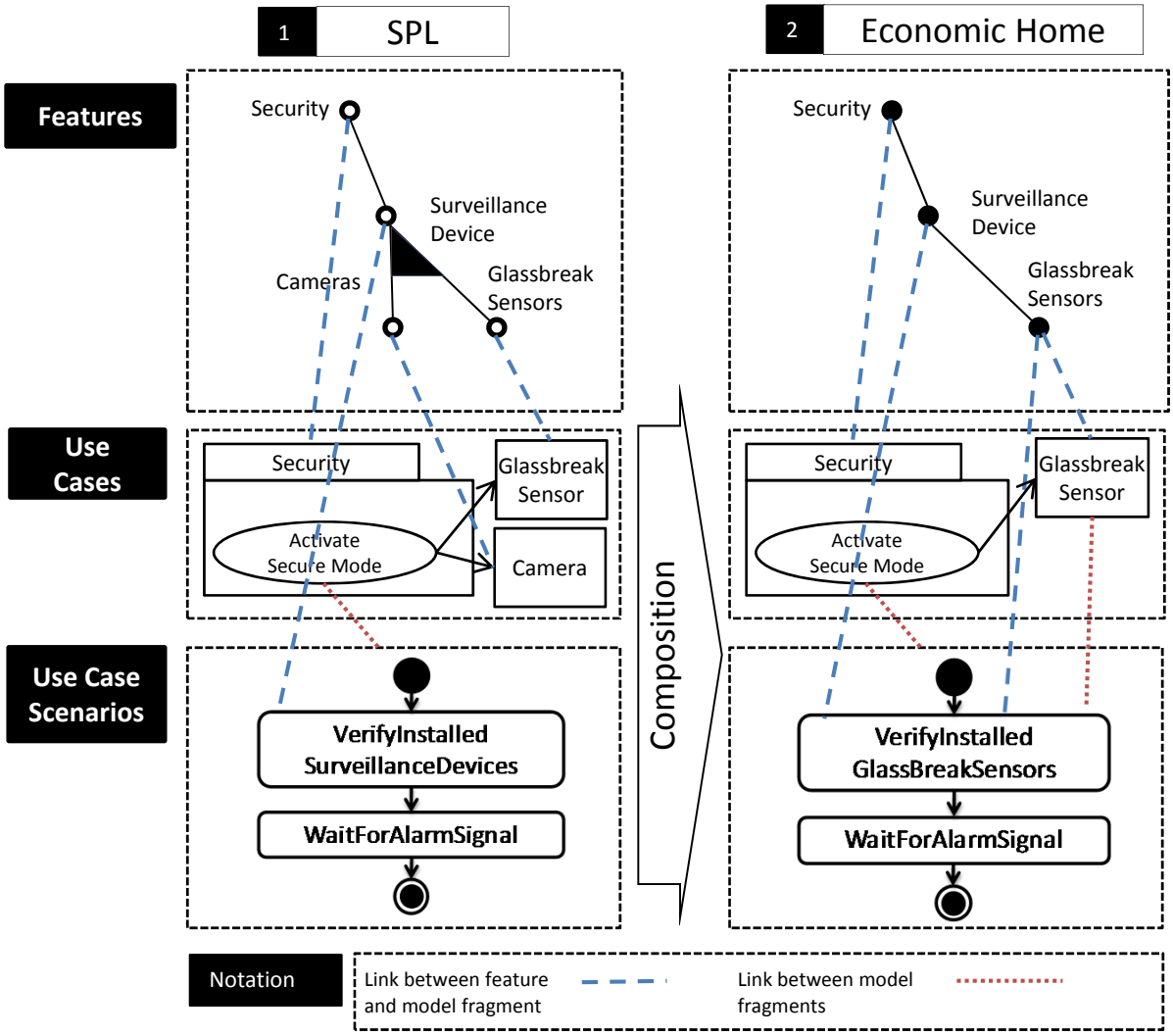


Figure III.5. Simplified Smart Home *ActivateSecureMode* use case scenario before and after a replace activity action.

Loughran, Fuentes, & Garcia, 2008) that difficult the use MDD techniques use by non technical-oriented people.

- *Testing and understanding the behavior of specific products in the SPL*: The automatic derivation of requirements models for a specific product is useful to both (i) understanding which requirements and features are involved in the development of an SPL product, and (ii) to support the testing and documentation activities. In particular, activity models are an example of requirements artifacts that are well suited for business process modeling and for modeling the logic captured by a single use case or scenario.

V. Future Research Directions

There are several issues to address that are common for model-driven specification approaches not only in requirements analysis, but also for architectural and detailed design, for example: co-evolution management of multiple interrelated models, verification and validation of the produced models, and usability of composition language. However, there are two main research directions in requirements specification. The first is related to the representation of non-functional requirements, business rules, and the use of specific requirements models for systems engineering; and the second is related to the detection of potential unwanted feature interactions analyzing the semantic of the composed models.

To specify interactions between functional and non-functional requirements during requirements modeling, other models, such as goal models (Chung, Nixon, Yu, & Mylopoulos, 1999; Yu) can be used. Such models also allow studying the actors and their dependencies, thus encouraging a deeper understanding of the business process. Therefore, we plan to use these kinds of models to specify also non-functional properties of SPLs systems. Besides, it is important to investigate the relationship of the models used in our approach and OMG (Object Management Group (OMG)) standards for business modeling such as: Business Motivation Model (BMM) (Object Management Group (OMG), 2008a), Semantics of Business Vocabulary and Rules (SBVR) (Object Management Group (OMG), 2008b), and also for systems engineering, such as

```
01 import features <''/featureModel.fmp''>;
02 import core <''/UseCaseAndActivityModels.uml'>;
03 concern SmartHomeSPL { //...//
04 variant for ElectronicWindows {
05   remove (useCase OpenWindowsAutomatically);
06   remove (actor WindowsActuator);
07 }
08 variant for and (glassbreak, not (cameras)){
09   replace (activity activateSecureMode::
10       VerifyInstalledSurveillanceDevices by
11       activity verifyInstalledGlassBreakSensors);
12 }
13 variant for not (security){
14   remove (package security);
15   remove (actor glassbreakSensor);
16   remove (actor camera);
17 }
18 variant for (GlassbreakSensor) {
19   insert (actor GlassbreakSensor);
20   insert (UCLinks_of_type: associatedWith {
21           from actor GlassbreakSensor to
22           useCase ActivateSecureMode} )
23   }
24 }
25 variant for (cameras){
26   insert (actor camera);
27 }
28 variant for not (cameras){
29   remove (actor camera);
30 }//...//
31 }
32
```

Table III
PART OF THE VML4RE MODEL FOR THE SMART HOME.

the requirements package of the Systems Modeling Language (SysML) (Object Management Group (OMG), 2010).

Another future research trend is the detection of feature interactions after the composition of their respective models. A feature interaction occurs when the behavior of a feature inhibits or subverts the behavior of another one in an unwanted or unexpected way. Therefore, feature interaction detection contributes to produce products whose features interact as expected and without conflicts (Alférez, Moreira, et al., 2009).

VI. CONCLUSION

This chapter provided an overview of both textual and graphical notations for modeling SPL requirements and different ways to compose requirements. Also, it shows our approach for model-driven requirements specification for SPLs. It separates requirements specifications, variability information and configuration knowledge to reach a better understandability of the models. This approach employs use cases whose detailed scenarios are modeled using activity models. The further elaboration of use cases with activity models; in contrast to free-format textual descriptions, facilitates to have models suitable to be processed by MDD tools and to produce useful information such as the customized models for specific products in a SPL.

ACKNOWLEDGMENTS

This work was partially supported by the European Project AMPLE, contract IST-33710, and the grant SFRH/BD/46194/2008 of Fundação para a Ciência e a Tecnologia, Portugal.

REFERENCES

Alexander, I., & Maiden, N. (2004). Scenarios, Stories, Use Cases. Chichester, UK: Wiley.

Alférez, M., Kulesza, U., Sousa, A., Santos, J., Moreira, A., Araújo, J., et al. (2008). A Model-Driven Approach for Software Product Lines Requirements Engineering. Paper presented at the 20th International Conference on Software Engineering & Knowledge Engineering, Redwood City, CA.

Alférez, M., Moreira, A., Kulesza, U., Araújo, J., Mateus, R., & Amaral, V. (2009). Detecting Feature Interactions in SPL Requirements Analysis Models. Paper presented at the 1st International Workshop on Feature-Oriented Software Development, Denver, Colorado.

Alférez, M., Santos, J., Moreira, A., Garcia, A., Kulesza, U., Araújo, J., et al. (2009). Multi-View Composition Language for Software Product Line Requirements. Paper presented at the 2nd Int. Conference on Software Language Engineering, Denver, USA.

Alves, V., Gheyi, R., Massoni, T., Kulesza, U., Borba, P., & Lucena, C. (2006, 2006). Refactoring Product Lines. Paper presented at the Proceedings of the 5th International Conference on Generative Programming and Component Engineering, Portland, Oregon, USA. AMPLE. (2009).

Ample Project. http://www.ample-project.net/

Antkiewicz, M., & Czarnecki, K. (2004). FeaturePlugin: Feature Modeling Plug-in for Eclipse. Paper presented at the 2004 OOPSLA workshop on eclipse technology eXchange.

Bonifacio, R., & Borba, P. (2009). Modeling Scenario Vari-

ability as Crosscutting Mechanisms. Paper presented at the Aspect Oriented Software Development.

Bragança, A., & Machado, R. J. (2007). Automating Mappings between Use Case Diagrams and Feature Models for Software Product Lines. Paper presented at the 11th International Software Product Line Conference.

Chung, L., Nixon, B., Yu, E., & Mylopoulos, J. (1999). Non-Functional Requirements in Software Engineering (1 ed.). Amsterdam: Kluwer Academic Publishers.

Clements, P., & Northrop, L. M. (2002). Software Product Lines: Practices and Patterns. Boston: Addison-Wesley. Cockburn, A. (2001). Writing Effective Use Cases. Boston: Addison-Wesley.

Czarnecki, K., & Antkiewicz, M. (2005). Mapping Features to Models: A Template Approach Based on Superimposed Variants. Paper presented at the 4th International Conference on Generative Programming and Component Engineering.

Czarnecki, K., & Eisenecker, U. W. (2000). Generative Programming: Methods, Tools, and Applications. New York: ACM Press/Addison-Wesley Publishing Co.

Czarnecki, K., Helsen, S., & Eisenecker, U. (2004). Staged Configuration Using Feature Models. Paper presented at the Third International Conference in Software Product Lines, Boston, Massachusetts, USA.

Deursen, A. v., & Klint, P. (2001). Domain-Specific Language Design Requires Feature Descriptions. Journal of Computing and Information Technology, 10. DSM forum website. (n.d.).

DSM Forum: Domain-Specific Modeling. http://www.dsmforum.org/

Eclipse Foundation. (n.d.). MDT-UML2Tools. http://www.eclipse.org/uml2/

Eriksson, M. (2006). An Approach to Software Product Line Use Case Modeling. Unpublished Doctoral Degree, UMEÅ University, UMEÅ, Sweden. Eriksson, M., Börstler, J., & Borg, K. (2005). The PLUSS Approach - Domain Modeling with Features, Use Cases and Use Case Realizations. Paper presented at the 9th International Conference on Software Product Lines.

Figueiredo, E., Cacho, N., Sant'Anna, C., Monteiro, M., Kulesza, U., Garcia, A., et al. (2008). Evolving software product lines with aspects: an empirical study on design stability. Paper presented at the 30th International Conference on Software Engineering (ICSE 2008).

Filman, R. E., Elrad, T., Clarke, S., & Aksit, M. (2004). Aspect-Oriented Software Development: Addison-Wesley.

Gomaa, H. (2004). Designing Software Product Lines with UML: From Use Cases to Pattern-Based Software Architectures. Reading, MA: Addison-Wesley.

Grzegorz, R. (Ed.). (1997). Handbook of Graph Grammars and Computing by Graph Transformation: Volume I. Foundations. River Edge, NJ: World Scientific Publishing Co., Inc.

Heidenreich, F. (2010). FeatureMapper: Mapping Features to Models. http://featuremapper.org/

Jacobson, I. (1992). Object-oriented Software Engineering: A Use CASE Approach. Reading, MA: Addison Wesley.

Jouault, F., & Kurtev, I. (2005). Transforming Models with ATL. Paper presented at the Model Transformations in Practice Workshop at MoDELS 2005. Kang, K., Cohen, S., Hess, J., Novak, W., & Peterson, A. (1990). Feature-Oriented Domain Analysis (FODA) Feasibility Study (Technical report, CMU/SEI-90-TR-021): Software Engineering Institute, Carnegie Mellon University.

Markovic, S., & Baar, T. (2005). Refactoring OCL Annotated UML Class Diagram. Paper presented at the International Conference On Model Driven Engineering Languages And Systems.

Moreira, A., Rashid, A., & Araújo, J. (2005, 2005). Multi-Dimensional Separation of Concerns in Requirements Engineering. Paper presented at the Proceedings of the 13th IEEE International Conference on Requirements Engineering, France.

Morganho, H., Gomes, C., Pimentão, J. P., Ribeiro, R., Grammel, B., Pohl, C., et al. (2008). Requirement Specifications for Industrial Case Studies (AMPLE Project, Deliverable D5.2).

Object Management Group (OMG). Object Management Group. http://www.omg.org/

Object Management Group (OMG). (2008a). Business Motivation Model. http://www.omg.org/spec/BMM/1.0

Object Management Group (OMG). (2008b). Semantics of Business Vocabulary and Business Rules (SBVR). http://www.omg.org/spec/SBVR/1.0/PDF

Object Management Group (OMG). (2010). Systems Modeling Language (SysML). http://www.omg.org/spec/SBVR/1.0/PDF OMG.

OMG's (2009a). Meta Object Facility (MOF). http://www.omg.org/mof/

OMG. (2009b). Unified Modeling Language. from http://www.uml.org/

OpenArchitectureWare.org. (n.d.). openArchitectureWare. http://www.openarchitectureware.org/

Pohl, K. (2006). Panel on Product Line Research: Lessons Learned from the last 10 years and Directions for the next 10 years. In 10th International Software Product Line Conference (SPLC 2006). http://www.sei.cmu.edu/splc2006/SPLC06-ResearchPanel-KP.pdf

Pohl, K., Böckle, G., & van der Linden, F. (2005).

Software Product Line Engineering: Foundations, Principles and Techniques. Berlin, Germany: Springer.

Sánchez, P., Loughran, N., Fuentes, L., & Garcia, A. (2008). Engineering Languages for Specifying Product-derivation Processes in Software Product Lines. Paper presented at the 1st International Conference on Software Language Engineering (SLE).

Taentzer, G. (2003). AGG: A Graph Transformation Environment for Modeling and Validation of Software. Paper presented at the 2nd International Workshop on Applications of Graph Transformations with Industrial Relevance (AGTIVE), Virginia, USA.

Volter, M., & Stahl, T. (2006). Model-Driven Software Development. Glasgow, UK: Wiley.

Yu, E. (n.d.). i* an Agent-oriented Modelling Framework. http://www.cs.toronto.edu/km/istar/

KEY TERMS & DEFINITIONS

Software Product Line (SPL) engineering: It is a development approach to increase software quality and productivity. It addresses the creation and management of a family of software products for a particular domain instead of developing each product separately.

Feature: It is a property or functionality that is relevant to some stakeholders and that allows to distinguish between products in a SPL. In a family of software products most of the features are common while some features vary between the family members.

Common feature: It is a kind of feature that expresses a property that is common to all the products in a SPL.

Variable feature: It is a kind of feature that expresses a property that is not common to all the products in a SPL.

Feature model: It is a model for expressing requirements in a domain on an abstract level. They are applied to describe variable and common features of products in a product line, and to derive and validate configurations of software systems.

Optional feature: It is a kind of feature that may not be included in some of the products of the SPL.

Variation point: It identifies a particular concept within the SPL requirements specification as being variable and it offers a number of variants.

Variant: It describes a particular variability decision, such as a specific choice among alternative variants.

Model Composition: It means to combine two or more models to modify one or more models, or to create a new one. Composition in SPL helps to create models for specific SPL products using composition actions and model transformations.

12

Evaluating Approaches for Specifying Software Product Line Use Scenarios

Authors: Mauricio Alférez, Rodrigo Bonifacio, Leopoldo Teixeira, Paola Accioly, Uirá Kulesza, Paulo Borba, Ana Moreira, João Araújo.

Paper Summary: This paper presents an empirical evaluation of four key and representative approaches (PLUSS, Model Templates, MSVCM and VML4RE) for specifying commonalities and variabilities of SPLs using scenarios. First, this paper proposes a metrics suite that takes into account relevant quality attributes for variability management, such as modularity, expressivity and stability, that aim at a better management of common and variable use scenario specifications between products of an SPL. Then, this paper provides details about the four approaches, the study settings, the CCCMS case study and the evaluation results. The details provided by this paper complement the evaluation of the DCC4SPL (represented by VML4RE) related to product-specific models derivation (Chapter 4 - Validation).

Authors Contribution: This research was the result of the work of a team of researchers from four universities. The team members that lead most parts of the paper were Rodrigo Bonifacio and I. The rest of the authors gave interesting comments on each version of the paper, collected results, and prepared model-based specifications for the evaluated approaches. Each author according to its experience and function helped to improve the content of the paper.

Publication Arena: This paper is under revision by an international journal.

Evaluating Approaches for Specifying Software Product Line Use Scenarios

Mauricio Alférez[a,*], Rodrigo Bonifácio[b], Leopoldo Teixeira[c], Paola Accioly[c], Uirá Kulesza[d], Paulo Borba[c], Ana Moreira[a], João Araújo[a]

[a] *CITI/Departamento de Informática, FCT, Universidade Nova de Lisboa, Caparica, Portugal*
[b] *Computer Science Department, University of Brasília Brasília, Brazil*
[c] *Informatics Center, Federal University of Pernambuco Recife, Brazil*
[d] *Computer Science Department Federal University of Rio Grande do Norte Natal, Brazil*

Abstract

A number of requirements engineering approaches have been proposed to specify commonalities and variabilities of software product lines using use scenarios. The existing assessments of these approaches are too informal, do not integrate different relevant quality attributes and derive conclusions by comparing just a couple of techniques. This paper presents an empirical evaluation that takes into account relevant quality attributes for variability management, such as modularity, expressivity and stability. It compares four key, representative techniques that aim at a better management of common and variable use scenarios specifications between products of an SPL. The techniques chosen to be evaluated span from type of notation (textual or graphical-based), style to resolve variability (based on annotations or compositions), and quantification expressiveness. The main contributions of this paper are a comparative study, the definition of a metrics suite, and the discussion on the design issues of the evaluated approaches for specifying variabilities in SPLs.

Keywords: Software Product Line Engineering, Variability Modelling, Use Scenarios, Variability Modelling, evaluation, Requirements Specification.

1. Introduction

Software Product Lines (SPL) have been pointed out, by current research and practical experience, as the way to achieve significant progress in software reuse [1, 2]. An SPL is defined as a family of "software-intensive systems that share a common, managed set of features that satisfy the specific needs of a particular market segment or mission and that are developed from a common set of core assets in a prescribed way" [2]. In this definition, a "feature" can be defined as "a system property that is relevant to some stakeholders and is used to capture commonalities or discriminate among systems in a family of software systems" [3]. Features that are common to all the SPL products are known as "commonalities" and those that are optional and alternative are known as "variabilities". Also, "variability" of an SPL is understood as the commonalities and differences between products in terms of requirements, architecture, components, and test artifacts [2].

Most work on SPL variability focuses on the design and code levels. However, variability management during SPL requirements engineering is essential to support requirements reuse [4]. Nonetheless, it is also useful when considering the effects of adding new requirements and removing others. Moreover, it can help when requirements are not fully understood and should be examined by stake-

*Principal corresponding author
Email addresses:
mauricio.alferez@campus.fct.unl.pt (Mauricio Alférez), rbonifacio@cic.unb.br (Rodrigo Bonifácio), lmt@cin.ufpe.br (Leopoldo Teixeira), prga@cin.ufpe.br (Paola Accioly), uira@dimap.ufrn.br (Uirá Kulesza), phmb@cin.ufpe.br (Paulo Borba), amm@fct.unl.pt (Ana Moreira), ja@fct.unl.pt (João Araújo)

holders that do not necessarily know about technical development issues such as managers and domain experts.

Use scenarios [5] are a recurring technique used in SPL engineering that is crucial to understand SPL features. A "use scenario", "usage scenario", or "scenario" for short[1], describes a real-world example of how one or more actors (e.g., users, organizations and sub-systems) interact with a system, describing the steps (i.e., events and/or actions) that occur during the interaction [5]. Scenarios can be composed according to specific combinations of features and, therefore, they are also useful to specify the intended behavior of target products of an SPL [6]. They are valuable assets as they provide examples of the system usage, therefore, playing an important role to both design and subsequent usability testing [5]. Such examples of system usage help different stakeholders, from managers and programmers to testers and domain experts, to understand and develop software products that satisfy their requirements.

Variability management is still a challenge for SPL engineering, largely due to its crosscutting nature that is evident along all core assets — ranging from documentation and scenarios to code-based modules and test cases [2, 7, 8]. In scenario specifications this challenge manifests mainly because: (i) the specifications of variable features often scatters through several scenarios; (ii) scenarios specifications are often tangled with information on how to configure the scenarios according to the presence or absence of different features. These two factors make it difficult to control composition of scenarios consistently along the whole specification and also hampers to modularize and reason about each concern of the system separately. To address this challenge, several approaches have applied aspect-oriented techniques [9, 10] for modeling scenario variability for SPLs (e.g., MSVCM [11] and VML4RE [12]). These can also be classified as compositional-based approaches in SPL terminology [13]. However, not all compositional-based approaches are necessarily aspect-oriented.

This paper assesses and compares four different approaches to specify scenarios variability in SPL against a set of releases from a common case study: the Car Crash Crisis Management System SPL [14] (Section 3). Our study takes into account three quality attributes (modularity, stability and expressiveness) that reveals differences among the four techniques and that have not been addressed by existing assessments [11, 15]. Therefore, the main contributions of this paper are:

- A comparative study of representative approaches for specifying scenario variability in SPLs. This considers aspect-oriented (AO) and non-aspect-oriented approaches which have different notations (textual or graphical-based) and different variability representation mechanisms (compositional or annotative). (We introduce them in Section 4).

- The definition of a metrics suite to compare existing SPL requirements specification approaches (Section 5.2). Although our metrics suite is partially adapted from existing aspect-oriented metrics, it also defines new metrics for quantifying expressiveness and stability of the relationships (also referred as mapping or configuration knowledge) between features and other artifacts (e.g., scenarios) [3]) in requirements specifications.

- A discussion on the design issues of the evaluated approaches for specifying variabilities in SPLs (Sections 6 to 8).

The result of this study concludes that aspect-oriented approaches reduce scattering of features and tangling of scenarios (Section 6); and improve expressiveness (Section 7) and stability (Section 8) of the specifications. Finally, we discuss potential threats to the validity of our findings in Section 10. Section 11 relates our work with other research topics and Section 12 presents our concluding remarks.

2. Goal and Research questions

The goal of this work is to analyze representative techniques for specifying SPL variability in scenarios, for the purpose of understanding their differences and the implications of their characteristics with respect to modularity of the features specifications (how localized is each feature specification along the scenarios), stability (effectively, how changes are required to evolve an SPL from one configuration to another), and expressiveness (how verbose the specification of those changes are), from

[1] From now on we mostly employ "scenarios" to refer "use scenarios".

the point of view of software engineers in the context of a common case study proposed by the SPL research community.

Taking into account this goal, there are three questions and correspondent attributes addressed in this work (Section 5 gives more details and relates the questions to the metrics suite):

> Question 1: Which and to what extent the techniques benefit a better modularization of feature specifications?
>
> Question 2: Which and to what extent the techniques benefit stability of their specifications (the scenarios, composition and configuration knowledge) after SPL evolution?
>
> Question 3: Which and to what extent the techniques benefit the expressiveness after SPL evolution?

The next two sections give an overview of the CCCMS SPL that we use to perform our study, and the four representative techniques for specifying SPL variability in scenarios.

3. Overview of the Car Crash Crisis Management System

In our study, we compare different specifications for the Car Crash Crisis Management System (CCCMS) [14]. A crisis is an unpredictable situation that can lead to severe consequences if not dealt with quickly. A car crash crisis management system facilitates the process of identifying, assessing, and handling the crisis situation by orchestrating the communication between all parties involved in handling the crisis, allocating resources and providing information to determined users. Any car crash crisis management system has a common set of responsibilities and functionalities. It is therefore natural to build an SPL of car crash crisis management systems, which can be specialized to create a particular kind of crisis and a particular context. The CCCMS case study became a benchmark in SPL modeling, their models are public and described in detail in [16] and [14]. The CCCMS involves single or multiple vehicles, and is limited to the management of human victims.

In [16], there are various non-aspect-oriented models that served as basis for our study, such as a feature model, use scenario textual descriptions, textual requirements, and a domain model. From those models, the feature model is the one that relates more directly to the SPL configurability. It identifies and relates common and variable features between the products in the domain of the SPL. Feature models depict a hierarchical decomposition of features with mandatory (must have if all its ancestors are selected), inclusive or (selection of zero or more), exclusive or (selection from many) and optional (may or may not have) relationships between features. A domain model consists of class diagrams that show the relationships between the main concepts of a domain.

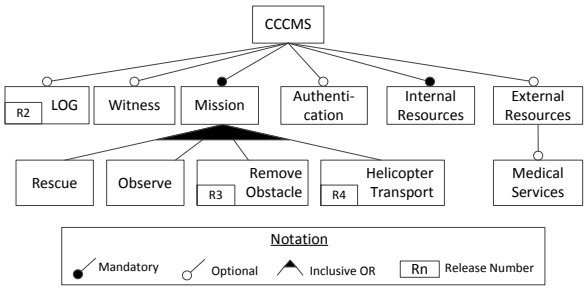


Figure 1: Partial feature model for the CCCMS SPL.

Figure 1 shows the part of the feature model of the CCCMS SPL that we considered in our evaluation, where 240^2 possible different combinations of features can be chosen. Figure 1 also represents, in the boxes on the bottom left of some features, the release (R2, R3, and R4) in which some features were introduced, since they were not part of the first release.

To resolve a crisis, the coordinator requests the employees and external resources to execute appropriate missions. A crisis is triggered when a witness places a call to the crisis center and is answered by a coordinator. The coordinator captures the witness report in the system (Witness feature), which recommends to coordinator the missions that have to be executed based on the current information about the crisis and resources (Mission feature). The coordinator selects one or more missions recommended by the system— for example, rescue (Rescue feature), observe (Observe feature), order a helicopter transport (Helicopter Transport feature) or remove obstacles (Remove Obstacle feature); these missions can be executed more than once and in parallel,

[2] An editable version of the feature model and statistics such as the number of valid product configurations are available in the feature model repository and online editor at http://www.splot-research.org/

if necessary. Then, the coordinator, depending on the mission, assigns internal resources (Internal Resources feature) or requests external ones (External Resources feature) to fulfill the mission. After communication between the resources and the coordinator, other information and new missions or resources can be called. Finally, all resources submit a final mission report so that the coordinator can finalize the crisis resolution process.

During missions, if medical services (Medical Services feature) are available in the system, CCCMS employees such as first aid worker can request to the system the victim's medical history information relevant to his injury from all connected hospital resource systems. Also, if available in the CCCMS, it will be possible to log all processes and decisions taken (Log feature). Similarly, while assigning internal resources, authentication in the system, or when taking important decisions, if available, it is necessary to request login and authentication in the system (Authentication feature).

We present more details on the specification and evolution of the CCCMS in Section 5.

4. Overview of the Evaluated Approaches

For this study we have chosen four approaches proposed for modeling variabilities in SPL requirements specifications: PLUSS, Model Templates, MSVCM, and VML4RE. These were chosen due to several reasons. First, they are representative from existing and recently proposed SPL requirements modeling approaches. Second, due to our knowledge and experience with them, as we wanted to fully explore the best characteristics of the selected techniques. Finally, because they adopt different composition and annotation based mechanisms, which we were interested in confront and compare to answer to our research questions. There are other approaches that can be grouped inside the previous mentioned categories and are very similar to the ones presented in this evaluation. For example, approaches such as PLUC [17] and PLUS [18] have inspired other works, but they were not taken into consideration in this study because more recent techniques such as PLUSS [6] and Model Templates [19] built upon them and provided more details in their application.

As stated previously, each approach represents different breeds of work that can be distinguished according to some design dimensions, such as the variability representation style and the specification notation. To address the crosscutting nature in the variability management, MSVCM [11] and VML4RE [12, 20] apply aspect-oriented techniques for modeling scenario variability for SPL. These can also be classified as compositional-based approaches in SPL terminology [13]. These approaches use independent models to express the relationships between specific fragments of the scenarios and SPL features. This relationship (also referred as mapping) between features and other artifacts (e.g., scenarios) is usually designated as "configuration knowledge" [3].

Other approaches, such as Model Templates [19] and PLUSS [6], are annotation-based [13] and differ from the aspect-oriented approaches in determining which and how specific parts of the scenarios are composed according to specific selections of features. In these approaches, variability in scenario specifications can be defined by adding annotations. These annotations are usually specified using a mapping table that relates specific parts of the scenarios to SPL features, or using UML stereotypes or notes with the feature name inserted directly on specific elements of the scenarios specifications. These annotations are placed throughout the specifications to determine which fragments of the scenario specifications are related to features of the SPL. Therefore, these approaches do not separate common and variable scenario specifications and do not use a dedicated configuration knowledge model to dictate how to compose the scenarios according to specific feature selections.

There are different ways to express scenarios, for example using a black-box textual notation. This notation textually describes and relates actor inputs and system responses into two columns of tables. UML activity diagrams can be used as an alternative to model scenarios to provide a different concrete syntax based on diagrams. For instance, PLUSS and MSVCM specify variabilities in textual scenarios whereas Model Templates and VML4RE specify variabilities in scenarios using a graphical representation of the requirements such as activity diagrams.

Next subsections give an overview of the approaches investigated in our study. For each approach, we mention its variability mechanisms, composition process, that is, how they manage to derive a product of an SPL, and relate their mechanisms to three different types of requirements variability [21]:

- Variability in function: occurs when a particular function (detailed as scenarios) might exist in some products and not in others.
- Variability in control flow: occurs when a pattern of *user-system* interaction within a scenario varies from one product to another.
- Variability in data: corresponds to fine grained variations and occurs whenever two or more scenarios share the same behavior and differ in relation to the values of a specific concept.

4.1. PLUSS

PLUSS is a domain approach to manage variant behavior in use case models[3] [6]. Indeed, it consists of a customization for feature modeling and a particular notation for specifying variant behavior in textual scenarios. These scenarios detail the whole use case model of an SPL. Next we present PLUSS according to the following characteristics.

Variability representation mechanisms. Choosing a feature for a product of an SPL might trigger the "selection" of a complete scenario or some steps in a scenario, or even a use case that encompasses several scenarios. Therefore, in order to allow the representation of variabilities, existing use cases, scenarios and individual steps in the scenario requirements specification must be related manually to features in the feature model. This kind of relationship between feature and parts of the requirements specification is named "require". Table 4.1 shows an example of this kind of annotation in steps to represent variability; here, the Execute Rescue Mission scenario is related to the Rescue Mission feature. Also, steps 2 and 3 are related with the Medical Services feature. The remaining steps are commonalities. As such, we do not associate them with any particular feature. Variability in values along the specification such as numbers or names, is supported using parameters each one related to an alternative or optional feature in the feature model. This kind of relationship is named "instantiate". When deriving an SPL member specification, the value assigned to a parameter corresponds to the selected subfeature(s) that the parameter refers to.

[3]Use scenarios describe a single path of logic whereas use cases typically describe several paths (usually the basic course plus any appropriate alternate paths).

Composition process: The process of deriving products in PLUSS basically (i) filters optional use cases, scenarios and steps that are related to features not selected in a specific product and (ii) assigns the selected features to the related parameters. It is important to emphasize that domain engineers have to annotate the specification detailing which features are related to use cases, scenarios, steps, and parameters. Thus, there is no independent model relating the use case model to features.

In our comparative study, we annotate optional scenarios and steps to indicate their dependencies with specific features. For instance, Table 4.1 shows that the Execute Rescue Mission scenario requires the Rescue Mission feature. In other words, this scenario is only present in products configured with the Rescue Mission feature. Moreover, Steps 2 and 3 in Table 4.1 are also optional. They are only present in products that are configured with the Medical Services feature.

Supported Variability: PLUSS supports the three types of variability: function, control flow, and data. Variability in function and control flow are made possible by relating entire scenarios and specific scenario steps, respectively, to features in the feature model. However, it is limited to only one feature by scenario or step. Variability in data is possible by adding parameters in the specification that are then instantiated according to the feature selection.

4.2. Model Templates

Model Templates (MTs) use activity diagrams to specify scenarios. In fact, a model template is an annotated model expressed in the target notation defined by a metamodel [19, 22]. Thus, a model template could be specified either using UML diagrams or any other domain specific notation defined using Meta Object Facility (MOF).

Variability representation mechanisms: Domain engineers have to relate model elements to features, in order to compose specific scenarios for an SPL product. In case of activity diagrams, for example, the model elements that can be related to features are actions, transitions flows, start and final nodes. However, differently from PLUSS, which relates each individual asset to one specific feature, this relationship is more expressive in Model Templates, since model elements can be related to feature expressions, represented as propositional formulas involving features. For example, Figure 4.2 shows two examples of propositional formulas: «medical

Scenario: Execute Rescue Mission (SC07).

Description: The intention of the First Aid Worker is to accept and then execute a rescue mission that involves transporting a victim to the most appropriate hospital.

Related feature: Rescue Mission

Flow of events:

Code	Related Feature	User Actions	System Responses
1	-	First Aid Worker transmits injury information of victims to System.	System updates crisis record with the sent injury information.
2	Medical Services	First Aid Worker determines victim's identity and communicates it to System.	System requests victim's medical history from all connected Hospital Resource Systems.
3	Medical Services	Hospital Resource System transmits victim's medical information to System.	System notifies First Aid Worker of medical history of the victim which is relevant for his injury.
4	-	-	System instructs First Aid Worker to bring the victim to the most appropriate hospital.
5	-	First Aid Worker notifies System that he has dropped off the victim at the hospital.	-
6	-	First Aid Worker informs System that he has completed his mission.	-

Table 1: PLUSS specification of Execute Rescue mission.

`services»` and `«not medical services»`. The use of feature expressions increases expressiveness because it avoids the need of polluting feature models with the introduction of artificial features, such as *not medical services*.

Composition process. Composition of scenarios is based on the implicit or explicit mechanisms to keep or remove model elements in the activity diagram. The explicit mechanism occurs when parts of the model are included in a product specification because they are related to a feature expression that is satisfied by the product configuration. For example, in Figure 4.2, the activity "System requests victim's medical history from all connected hospital resource systems" will be kept in all the SPL products that contain the medical services feature. Differently, the implicit mechanism occurs when dependencies among model elements are not satisfied. For instance, if one transition points to one activity that is not selected for a specific product, it will be implicitly removed from the product specification.

The model template that specifies the "Execute Rescue Mission" use case is shown in Figure 4.2. This model template is based on annotated activity diagrams, as discussed in [19]. Note that some activities and transitions between activities are annotated with the Medical Services stereotype, as well as a transition labeled with "A" in the diagram of Figure 4.2. These elements only appear in products configured with the Medical Services feature. Similarly, the transition labeled with "B" has the "not Medical Services" stereotype, stating that it will only be present if a product is configured without the Medical Services feature. Therefore, in a similar fashion to PLUSS, Model Templates scatters the configuration knowledge concern, represented by annotated feature expressions throughout the requirements models, and also presents tangling of different concerns (i.e., features) inside some models. Furthermore, because of the complexity of maintaining a generally large model of the overall system, it is difficult to distinguish and maintain possible alternative flows and configure appropriately their related feature expressions. In our example of Figure 4.2, if we add only one optional feature, we almost duplicate the number of activities and also use the same stereotypes repetitively inside the specification. We will further discuss these problems in Sections 6 and 7.

Supported Variability. Model Templates supports variability in function and variability in control flow. Variability in function occurs when a whole scenario is annotated with a feature expression, whereas variability in control flow occurs when a specific activity is annotated with a feature expression. As discussed, this is not limited to mapping a scenario (or activity) to one single feature; instead it allows a "many-to-many" mapping between model elements and features. It does not support variability in data.

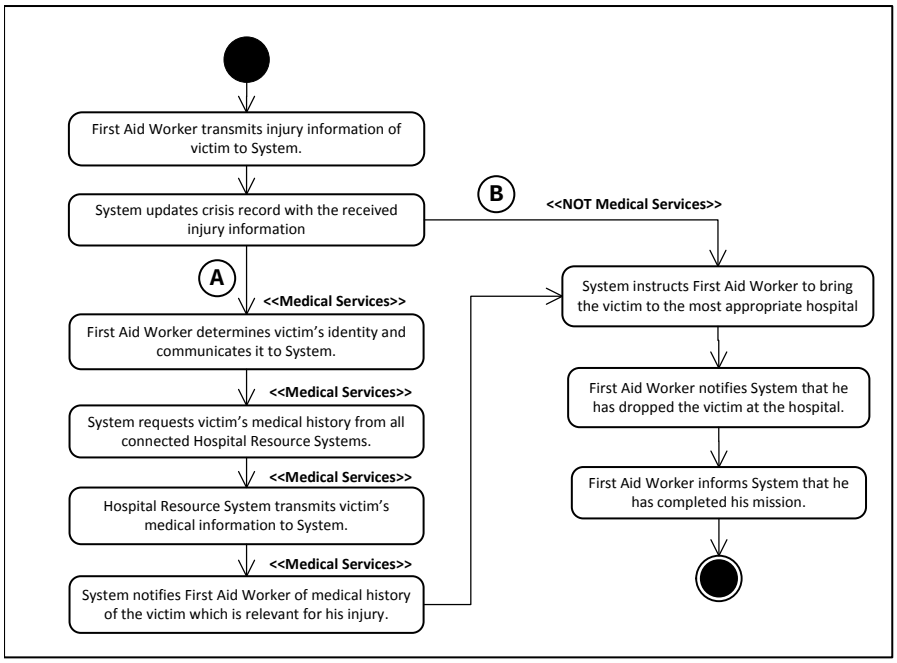


Figure 2: Model Template specification of Execute Rescue mission.

4.3. MSVCM

Similar to PLUSS, Modeling Scenario Variability as Crosscutting Mechanisms (MSVCM) is an approach to manage variant behavior using textual scenarios [11]. However, it has explicit and separated mechanisms to define variability and express configuration knowledge.

Variability representation mechanisms. To deal with variabilities between instances of a same scenario, MSVCM proposes new constructs to describe use cases: aspectual use cases and parameters. Using aspectual use cases makes it possible to change the behavior (represented as a sequence of steps) of an existing scenario. Scenario parameterization allows the configuration of scenarios that differ according to values in a specific domain.

Composition process: MSVCM aims at separating common from variant behavior. For instance, the scenario in Table 2 details the behavior required by the Rescue Mission feature. There is no step in this scenario related to the Medical Services feature, differently from the PLUSS specification depicted in Table 4.1. In MSVCM, the specification of the interaction between the aforementioned features can be modularized as an advice (see Table 3). Note that, in MSVCM, scenarios and advices do not make explicit references to features. Actually, an independent model, named configuration knowledge, is responsible for relating scenarios and advices to features. Presenting in more details, a configuration model relates feature expressions to transformations that translate SPL assets into product specific artifacts. If a feature expression is evaluated as True for a given product, the related transformations are applied.

Three distinct transformations are supported: (i) select scenario (includes a given scenario in the final product), (ii) evaluate advice (composes an advice through matched join points), and (iii) bind parameter (replaces parameterized textual sentences by feature data). Therefore, the configuration knowledge of Table 4.3 covers the configurability of the first release of the CCCMS SPL. Note that aspectual use cases in MSVCM support quantification. For instance, the advice ADV01 quantifies over all steps assigned to `@InjuryData` (see the *pointcut* clause of ADV01), which, differently from Model Template, does not require any transformation related to the *not Medical Services* feature expression.

Supported Variability. MSVCM transformations support the three types of variability previously discussed (variability in function, data, and control flow). For instance, the *select scenario* transforma-

Scenario: Execute Rescue Mission (SC07)

Description: The intention of the First Aid Worker is to accept and then execute a rescue mission that involves transporting a victim to the most appropriate hospital.

Flow of events:

Code	User Actions	System Responses
SC07.1	First Aid Worker transmits injury information of victims to System.	System updates crisis record with the sent injury information @InjuryData
SC07.2	-	System instructs First Aid Worker to bring the victim to the most appropriate hospital.
SC07.3	First Aid Worker notifies System that he has dropped off the victim at the hospital.	-
SC07.4	First Aid Worker informs System that he has completed his mission.	-

Table 2: MSVCM specification of Execute Rescue mission.

ADV01

Advice: Medical Service advising Execute Rescue Mission.

Description: Transmits injury information of victim to System.

Pointcut: @InjuryData

Flow of events:

Code	User Action	System Response
ADV01.1	First Aid Worker determines victim's identity and communicates it to System.	System requests victim's medical history information from all connected Hospital Resource Systems.
ADV01.2	Hospital Resource System transmits victim's medical history information to System.	System notifies First Aid Worker of medical history of the victim relevant to his injury.

Table 3: MSVCM specification of the Advice for Medical Services.

Expression	Transformations
CCCMS	select scenario SC01, SC03, SC04
Authentication System	select scenario SC10
Rescue Mission	select scenario SC07, SC08
Witness	select scenario SC02
Remove Obstacle Mission	select scenario SC09
Medical Services	evaluate advice ADV01
Observe Mission	select scenario SC06

Table 4: MSVCM configuration knowledge for release 1 of the CCCMS SPL.

tion supports variability in function, allowing us to select specific scenarios for a given product configuration. Differently, the *evaluate advice* transformation deals with variability in control flow (as required by the Medical Services feature that changes the behavior of the Execute Rescue Mission use case). Finally, the *bind parameter* transformation deals with variability in data, mapping parameters within the specifications to specific data obtained from the feature configurations.

4.4. VML4RE

The Variability Modeling Language for Requirements (VML4RE) [12, 20] presents a solution for the composition of model fragments for requirements models of an SPL which includes use case diagrams and their related scenarios represented by activity diagrams. This approach aims at specifying the composition of requirements models for specific products of an SPL using a separate composition model that contains transformations (named *actions*) specially-tailored for scenarios.

Variability representation mechanisms. VML4RE provides a set of specialized operators for referencing and composing parts of the scenarios and use case model. It does not add

annotations to the scenarios as happens with Model Templates, and it does not use free-format textual descriptions that can be ambiguous, due to the interpretation of the natural language [1]. Similar to MSVCM, it employs a separate model to specify configuration knowledge, called composition model. The composition model allows the specification of transformations (actions), which are linked to feature expressions.

Composition process. VML4RE composes specific use case models and scenarios according to the actions expressed in the composition model. Actions are responsible for different kinds of modifications in the models, such as insert, connect, merge, remove and replace use cases, actors, packages, activity diagrams, steps and their relationships.

Figure 3 shows the scenario Execute Rescue Mission (SC07) that details the behavior required by the Rescue mission feature, and Figure 4 shows the steps that will be merged to the Execute Rescue Mission activity diagram. The VML4RE code snippet in Figure 5 merges the Injury Data advice with the Execute Rescue Mission scenario. This code snippet is part of the composition model that represents configuration knowledge, as it relates the Medical Services feature to some parts of the scenarios. The merge action (Line 2) copies the elements referenced by the expression given in the second argument, the advice, to the model referenced by the expression given in the first argument, the base model. To avoid duplicated model elements identifiers, this operator adds a prefix (the name of the advice scenario) to the identifier of the new model elements. After copying the model elements, it is necessary to redirect the control flow to the steps included in the advice. This is done by adding new control flows (Lines 3 and 4) and removing an unnecessary control flow (Line 5).

Supported Variability. In VML4RE, variability in function and control flow are possible using a separate configuration knowledge model, which may include entire scenarios or specific scenarios steps respectively, according to a feature expression. Variability in data is also possible replacing generic activities or steps by more specific activities.

4.5. Summary

Table 5 summarizes the main characteristics of the techniques explained. The second column refers to the notation used to model scenarios. PLUSS and MSVCM use textual scenarios descriptions following a blackbox format. On the other hand, Model Templates (MT) and VML4RE use UML activity diagrams which employ a graphical notation.

The mechanisms to represent variability can be divided into two types: annotations and compositions. Annotation-based techniques introduce annotations on the scenarios to indicate variable parts. PLUSS and Model Templates keep or remove parts of the scenarios depending on the evaluation of their annotations according to specific product configurations.

Composition-based techniques model the variations as distinct modules and so, to generate the scenarios for an SPL member, there must be a composition of variable and common modules. VML4RE is a compositional approach because activity diagrams are composed to add, replace or remove parts of the initial base scenarios. MSVCM is considered to be both compositional and annotative. MSVCM is compositional because it uses different modules to represent commonality and variability (advices) but it also uses annotations in the base scenarios to show where the advices should be applied.

All the approaches support variability in function, control flow and data, except by Model Templates that does not have specific mechanisms to instantiate data inside the scenarios based on specific feature configurations.

5. Study Settings

This section presents detailed information about our comparative study of the SPL requirements engineering approaches just introduced. Section 5.1 presents the phases and assessment procedures of our comparative study. Section 5.2 describes the metrics suite adopted to enable the analysis of modularity, stability and expressiveness of the requirements specifications using the approaches investigated in our study.

5.1. Study phases and assessment procedures

Our study was organized in three major phases:

1. Specification of the Car Crash Crisis Management System SPL (CCCMS) using the four chosen requirements approaches.
2. Evolution of the specifications in the different approaches, in order to address the change scenarios.
3. Quantitative assessments of the different specifications and releases of the CCCMS SPL.

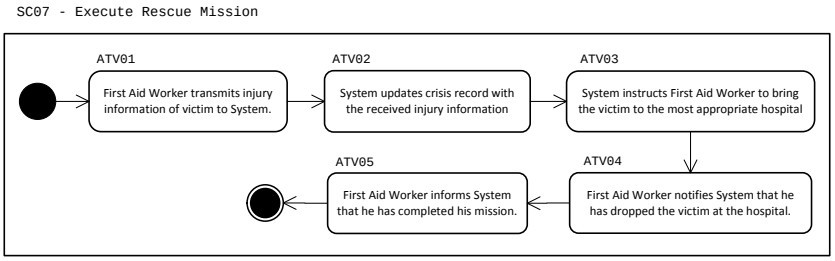


Figure 3: VML4RE specification of Execute Rescue mission.

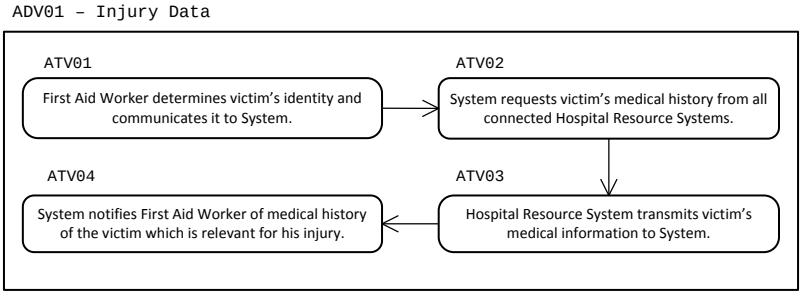


Figure 4: VML4RE specification of the Injury Data advice.

In the first phase, the CCCMS SPL was specified using the different modularization and composition mechanisms available in the investigated requirements approaches. From the models detailed in [14], we have developed a set of incremental releases for the CCCMS (they are available online[4]). Considering the feature model shown in Figure 1, we have defined a base release (R1), consisting of the features *CCCMS*, *Authentication System*, *Rescue Mission*, *Witness*, *Medical Services*, *Internal and External Resources* and *Observe Mission*. After that, all specifications were evolved to address the change scenarios corresponding to the releases R2—R4, that appear in Figure 1. The features inserted in the releases required the introduction of new scenarios and changes to existing ones. For example, R2 introduced the Log feature, which affects two existing scenarios: SC06-Execute Super Observer Mission and SC07-Execute Rescue Mission. However, in the subsequent releases (R3 and R4), the new use cases introduced (*Remove Obstacle Mission* and *Helicopter Transport Mission*) also had to deal with Log. These change scenarios allowed us to exercise the different modularization and composition mechanisms provided by each approach, to observe their modularity, stability, and expressiveness. Finally, we applied our metrics suite (see Section 5.2), to analyze and compare the obtained results for the different specifications.

All specifications were written according to alignment rules, which were necessary not only to verify that good practices were used in the approaches, but also to ensure that the comparison of the specifications was equitable and fair. Three researchers performed these alignment activities. All misalignments found were discussed between the study participants and eventual corrections were applied to the specifications to guarantee their alignment. For example, we ensured that: (i) every variability was modularized using the appropriate modularization and composition mechanisms of each approach; (ii) textual and graphic-based approaches used an equal number of elements that represents the same abstraction (such as activities in VML4RE and Model Templates, and textual steps for the scenarios in MSVCM and PLUSS approaches); and (iii) the specifications reflect the same functionalities/features and are consistent between them.

5.2. The metrics suite

Since we are interested in comparing different SPL requirements engineering approaches from the

[4] http://www.mauricioalferez.com/REJ/REJ-Data.htm

```
1.  Variant for (Medical Services) {
2.          merge ( "SC07" , "ADV01" );
3.          connect ( "SC07:: ATV02" , "SC07::ADV01_ATV01" );
4.          connect ( "SC07:: ADV01_ATV04" , "SC07:: ATV03" );
5.          removeControlFlow ( "SC07::ATV02" , "SC07::ATV03" );
6.  }
```

Figure 5: VML4RE code snippet to insert InjuryData advice.

Technique	Dominant Notation	Var. Mechanism	Supported Var.
MSVCM	Textual	Compositional-Annotative	F / CF / D
PLUSS	Textual	Annotative	F/ CF / D
VML4RE	Graphical	Compositional	F/ CF / D
MT	Graphical	Annotative	F / CF

Table 5: Summary of the main characteristics of the techniques. Notation: (F) Function, (CF) Control Flow, (D) Data.

modularity, stability, and expressiveness perspectives, we selected a metrics suite that allows the quantification of these attributes in the different specifications of the case study. Table 6 gives an overview of the metrics suite used in our study.

Our modularity investigation relies on two metrics that we have customized [25] from [23]: *degree of scattering of features* (DoS) and *degree of tangling of scenarios* (DoT). According to equations (1) and (2), DoS quantifies the concentration of a feature over each scenario $s \in S$ (the set of scenario specifications). Values of DoS are normalized between 0 (completely localized) and 1 (completely scattered). The greater the DoS of a feature f is, the greater is the probability of reviewing different scenarios when the specification of f has to evolve. Note in equation (1) that $\mid S \mid$ denotes the cardinality of the set S.

$$DoS(f) = 1 - \frac{\mid S \mid \sum_{s \in S} (Conc(f,s) - \frac{1}{\mid S \mid})^2}{\mid S \mid -1} \quad (1)$$

$$Conc(f,s) = \frac{number\ of\ steps\ in\ s\ assigned\ to\ f}{number\ of\ steps\ assigned\ to\ f} \quad (2)$$

Likewise, according to equations (3) and (4), DoT considers how many steps of a scenario are related to each feature $f \in F$ (the set of features). Values of DoT are similarly normalized between 0 (completely focused) and 1 (completely tangled). The greater the DoT of a scenario s is, the greater the probability of reviewing s when one of the related features changes. Usually we use the metric Degree of Focus (DoF) to refer to low tangling (DoT near to 0) in scenarios. DoF is easily derived from DoT and corresponds to $1 - DoT$. Thus, the lower the DoF of a scenario s is, the higher is the tangling (DoT) of features it specifies.

High values in the degree of focus and low values in the degree of scattering are usually associated to well-modularized systems [23, 26, 27].

$$DoT(s) = 1 - \frac{\mid F \mid \sum_{f \in F} (Dedi(s,f) - \frac{1}{\mid F \mid})^2}{\mid F \mid -1} \quad (3)$$

$$Dedi(s,f) = \frac{number\ of\ steps\ in\ s\ assigned\ to\ f}{number\ of\ steps\ of\ s} \quad (4)$$

Note that in equations (2) and (4) to evaluate these metrics we have to assign features to the individual steps of a specification. We follow the *configuration dependency analysis* as a guide [25], considering that a step *st* depends on a feature f if, and only if, the selection of f triggers the configuration of s. Similarly we assigned the modeled activities to features in the Model Templates and VML4RE approaches.

Regarding our stability assessment, we adapted a metrics suite that has also been validated and used to compare semantic and syntactic approaches for aspect-oriented requirements engineering [24]. These metrics quantify the stability of specifications and code elements that represent a software artifact in the context of evolutionary scenarios. In our study, we used them to quantify the stability of the requirements specifications along the different

Attribute	Metric
Modularity	Degree of scattering of features [23].
	Degree of focus of scenarios [23].
Stability of the specifications	Number of steps introduced or changed between two releases [24].
Stability of the compositions	Number of compositions items introduced or changed between two releases [24].
Stability of the CK	Number of configuration items introduced or changed between two releases.
Expressiveness of the composition	The ratio between the number of composition items and number of matched join points [24].
Expressiveness of the CK	Number of tokens required to specify the configuration knowledge.

Table 6: Metrics suite used in our study.

releases of the CCCMS SPL. We measured the stability of the specifications and the stability of the compositions. They are quantified by the number of modified or introduced steps or scenarios, and the number of modified or introduced composition items between two subsequent releases. In addition, we have also proposed a metric that quantifies the stability of the configuration knowledge.

Finally, we investigate the expressiveness of the specifications. For measuring the expressiveness of the configuration knowledge, we count how many tokens are required to map features to other models. For example, in PLUSS we have the "related feature" column that associates steps to features. Therefore, in the case of the PLUSS specification of Execute Rescue Mission shown in Table 4.1, we associate steps 2 and 3 with the Medical Services feature. Thus, the total count of tokens for this scenario is 4. We count tokens similarly for Model Templates, using the stereotype annotations. In the case of MSVCM and VML4RE, we count all tokens used in their respective dedicated configuration knowledge models (Subsection 7.1 provides concrete examples). Although the unit to quantify expressiveness of the configuration knowledge is rather low level, it allowed us to uniformly assess the different representations of the configuration knowledge. To measure expressiveness of the compositions, we use the notion of reachability [24], computing the ratio between the number of matched join points and the number of composition items. Finally, we only measure stability and expressiveness of compositions for the MSVCM and VML4RE compositional-based approaches. Since the other two approaches are annotation-based, variant behavior is already composed into specifications.

6. Assessment of Variability Modularity

This section presents our analysis of modularity regarding scattering of features and tangling of scenario specifications. Here we consider the different releases of the case study specified using each technique and, as mentioned before, we use a metrics suite that quantifies the degree of scattering of features (Section 6.1) and the degree of tangling and degree of focus of specifications (Section 6.2).

6.1. Degree of scattering of features

The degree of scattering (DoS) of a feature quantifies to what extent the specification of a feature is disperse. In our study, most of the feature specifications are well localized, which led to specifications with low DoS. Actually, only two features present some scattering: *Log* and *Authentication*.

The specification of Log imposes a homogeneous behavior that scatters throughout all use cases related to the Mission sub-feature (see Figure 1) in the case study. Differently, the Authentication feature requires two distinct procedures: (i) one related to the login behavior; and (ii) another one that verifies if an employee had already been authenticated. Using the compositional approaches (VML4RE and MSVCM), we could modularize these procedures in independent assets (advices). Nevertheless, such decision leads to the scattering of the Authentication feature specification, which could also be realized in the annotative based specifications. Figure 6 shows how we could separate these procedures using VML4RE. However, after considering other factors, such as the growth of the configuration knowledge, we decided to merge the complete specification of the Authentication feature in VML4RE (see Figure 7). In fact, this was a controversial decision, since merging both procedures in a single asset eliminates the Authentication feature

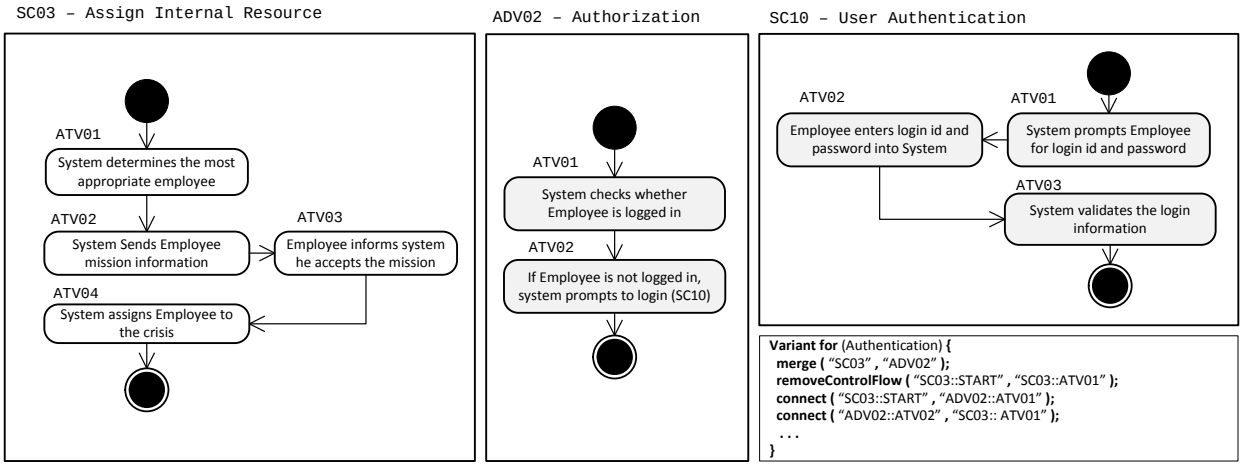


Figure 6: Separation of the Authentication feature in one Authorization advice (ADV02) and one Login scenario (SC10) in VML4RE. In this case, the Authorization advice could be merged into different scenarios.

scattering, even though it increases the coupling between the mentioned procedures, which hampers the possibilities to reuse the specific steps of the authorization procedure in other scenarios. Due to the small overhead on the configuration knowledge, we decided to specify the Authentication feature in MSVCM using a decomposition such as depicted in Figure 6.

For the first release (R1), Table 7 details the results of the feature assignment process, which relates features to the scenarios (and advice) steps. As explained in Section 5, the Log feature was not detailed in the first release of the CCCMS specifications. In that case, only the Authentication feature presents some scattering, leading to a DoS of 0.56 in MSVCM, Model Templates, and PLUSS specifications. Considering the other features, that were well modularized in the first version (leading to a DoS of zero), the resulting average DoS of those techniques was 0.07 (see Figure 8). Since we merged the specifications of the Authentication feature in VML4RE, all features were completely modularized in VML4RE— leading to an average degree of scattering of zero.

Tables 8 and 9 summarize the assignment of features to the specifications' steps of the last release. Since release R2, we could modularize the Log specification using both MSVCM and VML4RE. For this reason, introducing new scenarios in the later releases (R3–R4) of MSVCM reduced the average degree of scattering (leading to an average DoS of 0,05). Differently, the Log specifications increased

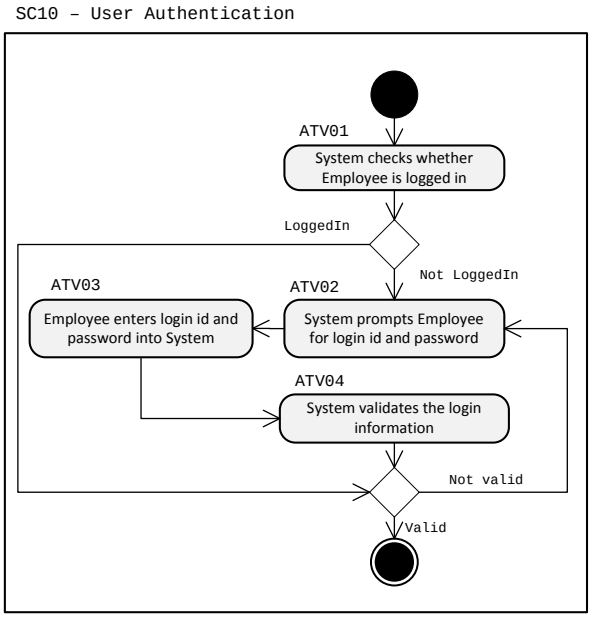


Figure 7: In this version, both Authorization and Login behavior are represented as a single scenario (SC10) in VML4RE. This version, although reducing the scattering of the Authentication feature, compromises the reuse of specific steps related to authorization.

the average degree of scattering in PLUSS as well as in Model Templates (DoS of 0,13 and 0,11 respectively). Actually, the DoS was even higher in PLUSS because the complete behavior of the Log feature was scattered throughout the missions specified using the PLUSS notation (See values of 5 from SC06 to SC09 in Table 9). In contrast, the Model Templates specification required basically one `Log Activity` in each mission, indicating the right point in the specification where the log behavior had to start (See values of 1 from SC06 to SC09 in Table 9). For this reason, we could say that the Log behavior in Model Templates was partially extracted from the missions and specified in a particular activity diagram (SCLog - Log Service).

To understand the behavior of a mission, for example, it would be necessary to compose the Log behavior with the existing mission specifications. Otherwise, we would not be able to reason about the complete specification of each mission. Therefore, without proper mechanisms for composing specifications, as supported by MSVCM and VML4RE, relating specifications by means of references could harm understandability, even though this design leads to a better modularization. Using PLUSS, we could have specified the Log behavior in a similar fashion as we have specified it using Model Templates, but the mentioned problem would also have arisen with PLUSS.

After quantifying the average DoS metric (Figure 8), we noticed that MSVCM and VML4RE reduce, or even eliminate, the scattering of features for the releases of the CCCMS SPL that we modeled. Differently, we are not able to eliminate the Log scattering in PLUSS and Model Templates specifications, mainly because they do not support the composition of common and variant behavior.

6.2. Degree of tangling and degree of focus of specifications

Degree of tangling (DoT) and degree of focus (DoF) (that corresponds to 1-DoT) measure how dedicated a scenario is to one or more features of the SPL. Figure 9 summarizes the corresponding average Degree of Focus (DoF) of the evaluated specifications. Note that there is no tangling (DoT=0) in the MSVCM and VML4RE specifications in the CCCMS, which leads to an average DoF equal to one in those techniques.

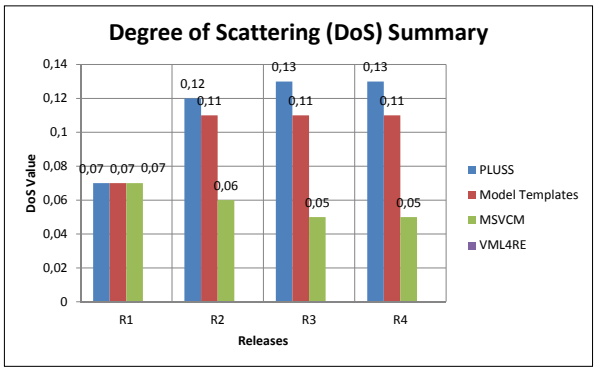


Figure 8: Average degree of scattering.

In contrast, we were not able to remove the tangling associated to the *Authentication* and *Log* features using either PLUSS or Model Template. This tangling occurs because:

- In both techniques (PLUSS and Model Templates), the Authentication behavior was specified within the specification of the scenario that assigns tasks to internal resources of the CCCMS. Therefore, using these techniques, the specifications regarding Authentication are tangled with the assignment tasks to internal resource specifications.

- Similarly, the specifications using both techniques tangle the Log behavior within the specifications of each mission— since all relevant information (such as used strategies, duration of resolution, and problems encountered) have to be registered for each assigned mission.

In fact, the resulting tangling was even higher in PLUSS, because the entire Log specification was scattered throughout several scenarios as it was previously explained. In a different way, we introduced just one activity related to the Log behavior in each mission specified using Model Templates.

The analysis of DoS and DoF here suggests that most of the features require localized and independent specifications. This is a different result when comparing with other studies that evaluated these metrics in source code. For instance, Marc Eaddy found that 95% of the concerns are scattered through the modular units of a source code [23]. Identifying why those findings were so different is a matter of future work.

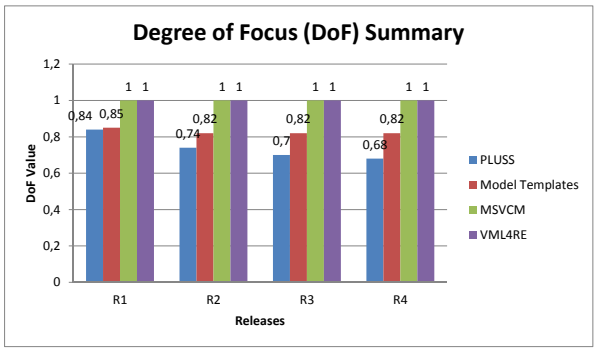


Figure 9: Average degree of focus.

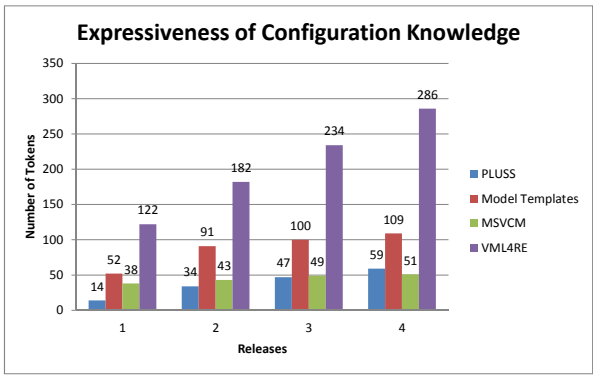


Figure 10: Expressiveness of the configuration knowledge.

7. Analysis of Expressiveness

This section presents our analysis of expressiveness considering the different releases of the case study for each technique. We used a metrics suite that quantifies the expressiveness of the configuration knowledge (Section 7.1) and the expressiveness of the compositions (Section 7.2).

7.1. Expressiveness of the configuration knowledge

In our study, the expressiveness of the configuration knowledge was measured in terms of tokens. In the case of PLUSS and Model Templates, we count tokens looking at the annotations on steps (PLUSS) and transitions between activities (Model Templates). For example, the annotation "`Medical Services`" has 2 tokens in PLUSS or Model Templates. For MSVCM and VML4RE, we count tokens looking at the specific models used to describe the compositions. For example, "`Medical Services evaluate advice ADV01`" in the MSVCM configuration knowledge model in Table 4.3 has 5 tokens. An example for VML4RE can be found in the composition model in Figure 5 lines one, two and six, where "`Variant for (Medical Services) {merge ("SC07" , "ADV01");}`" has 19 tokens.

Looking at the absolute numbers (Figure 10), we notice that graphics-based approaches Model Templates, and specially VML4RE, are more verbose than PLUSS and MSVCM to describe the configuration knowledge. PLUSS and Model Templates only associate features with variant behavior. MSVCM and VML4RE use dedicated models to describe all compositions, both common and variable.

If we observe the growth of these numbers, as illustrated in Figure 11, from the first release (R1) to the last (R4), we notice that it is greater in PLUSS (321%) than the others. The graphics-based approaches Model Templates (109.62%) and VML4RE (134.43%) grow in a similar rate. The growth rate for Model Templates could be a lot higher (comparable with PLUSS) if we had not used one separate diagram for the Log Services feature, as already discussed. Based on the collected measurements, we can argue that while composition-based approaches require a bigger effort to build a first version of a release (larger upfront investment) than annotation-based approaches (PLUSS and Model Templates), their evolution happens in smaller increments, by requiring a reduced number of new constructs. The large numbers for VML4RE indicates that its composition language could be improved using new actions and removing unnecessary syntactic sugar to reduce verbosity when specifying the configuration knowledge.

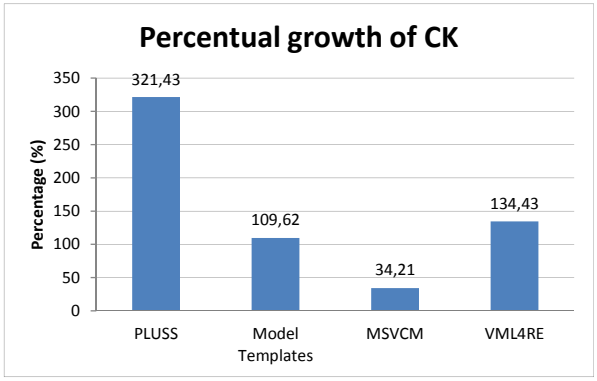


Figure 11: Growth of the configuration knowledge.

7.2. Expressiveness of compositions

From the four approaches under investigation, only MSVCM and VML4RE offer specific mecha-

nisms to compose common and variant behavior. To quantify the expressiveness of compositions, we use the notion of *reachability* [24], computed as the ratio between the number of matched join points and the number of composition items [24]. Composition items in MSVCM correspond to the pointcut clauses of advice. Therefore, MSVCM compositions are defined within the *specification language*, whereas composition items in VML4RE correspond to actions such as the *connect* construct detailed in each variant element of the VML4RE *configuration language*.

In our study, the reachability of VML4RE compositions is one— each VML4RE composition *reaches* a particular join point[5]. In contrast, MSVCM supports different mechanisms for quantification. For instance, the Log advice is applied after all scenarios named with the pattern `%mission%`. Consequently, the average expressiveness of the composition increases (see Figure 12) in the later releases of MSVCM specifications, since new missions are introduced.

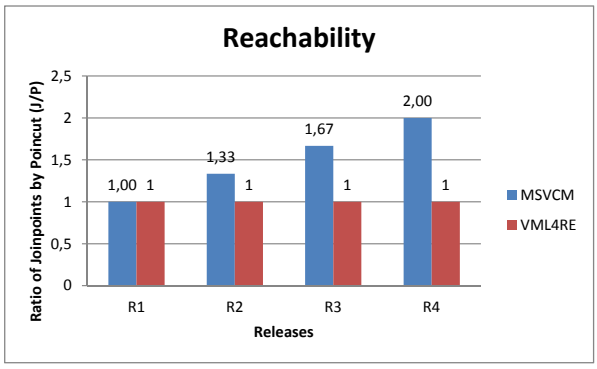


Figure 12: Average degree of reachability.

By comparing these results with the stability of composition assessments (Section 8.2), we could realize a strong correlation between the expressiveness and the stability of the compositions— the greater the stability, the greater the expressiveness of the compositions. Introducing new specifications that satisfy a composition item does not require changes in the composition concern. Besides that, a potential side effect regarding expressiveness is that undesired joinpoints might be caught by a composition item. In those situations, the composition item must be refined.

8. Analysis of Stability

This section presents our analysis of stability considering the different releases of the case study specified with each technique. We used a metrics suite that quantifies the stability of the specifications (Section 8.1), the stability of the compositions (Section 8.2), and the stability of the configuration knowledge (Section 8.3).

8.1. Stability of specifications

Figure 13 shows how many steps in the MSVCM and PLUSS approaches (or activities in the Model Templates and VML4RE approaches) have been introduced to evolve the specifications from one release to another. It can be noticed that more steps were introduced to evolve the annotative techniques (PLUSS and Model Templates), since the specification for the Log feature is not well modularized.

For instance, the second release introduced the Log feature, whose entire specification is scattered throughout the *Rescue* and *Observe* missions of PLUSS specifications. Since five steps were required to specify the Log feature, we had to introduce a total of ten new steps in the second release of the PLUSS specifications (five steps for each scenario that requires the Log behavior). In the Model Templates specification, one activity was introduced in each mission (to indicate where the Log behavior should start) and a new activity diagram for specifying the Log behavior was created. Using the compositional approaches (MSVCM and VML4RE), only steps for describing the Log behavior had to be created, in such a way that no additional steps were introduced in the original specifications of Rescue and Observe missions.

The latter releases (R3 and R4) detailed the behavior of Helicopter Transport and Remove Obstacle missions, which also require the log behavior. Besides the steps related to those missions, in PLUSS we had also to detail the steps of the Log behavior within both missions. For this reason, the number of introduced steps is higher in PLUSS than in the other approaches. For instance, in Model Templates we just had to specify the behavior of the new missions, plus one specific activity for indicating the point where the log behavior starts.

[5] An early version of VML4RE [28] had some language constructs that simplify matching specific fragments (i.e., join points) in the scenarios reducing the verbosity of the composition model. For example, pointcut designators such as "equal", "startsBy", "finishesWith", "contains", and quantifiers such as "*", "?".

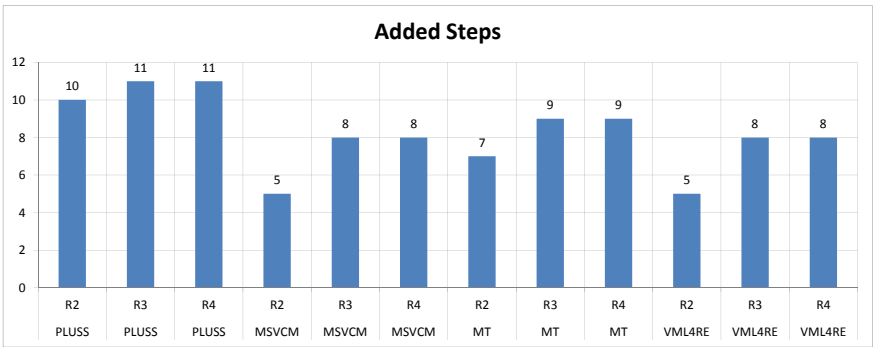


Figure 13: Number of steps introduced in each release.

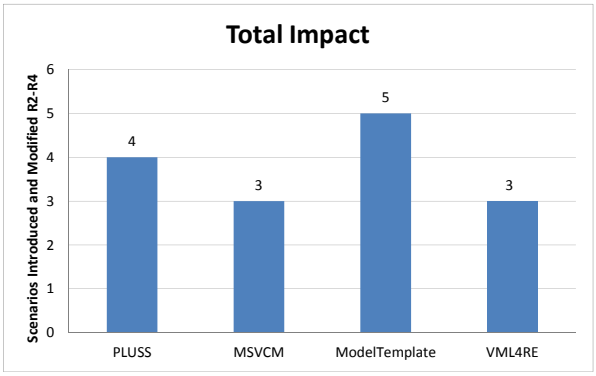


Figure 14: Stability of the specifications.

Moreover, we did not have to introduce additional steps using MSVCM and VML4RE, which reduced the number of introduced steps.

Also regarding the stability of specifications, Figure 14 summarizes the number of scenarios that have been introduced and modified, according to changes required to evolve the specifications from the first to the last release. In Figure 14 PLUSS and Model Templates required a higher number of introduced and modified scenarios. Indeed, the evolution of SPL specifications using the annotative style requires to take into account many places affected by the propagation of the changes.

8.2. Stability of compositions

We measured stability in terms of the number of composition items required to specify each CCCMS release. Since in PLUSS and Model Templates variability is tangled with core functionality, only MSVCM and VML4RE were evaluated. Remember that composition items in MSVCM are specified in the pointcut clause of advices, whereas compositions in VML4RE are specified in the variant construct from the textual description in that language.

In the first release, since each of the optional features (*Authentication* and *Medical Services*) changes a specific point of CCCMS, the number of composition items in both techniques is the same (Figure 15). In the second release, the Log behavior was specified in *one MSVCM advice whose composition item refers to all missions*. Differently, two composition items were required to indicate that Log advice in VML4RE should be applied to the Observer and Execute Rescue missions. For that reason, the number of composition items in MSVCM increased by one in the second release, while in VML4RE the number of composition items increased by two in the second release.

Likewise, introducing new missions that had to be advised by the Log behavior did not require new composition items in MSVCM. However, new composition items had to be introduced to *connect* the Log advice specified in VML4RE to the new missions specified in the third and fourth releases of the case study. This leads to a worse stability of the compositions, when compared to MSVCM. This highlights the benefits of using more expressive mechanisms to compose common and variant behavior.

8.3. Stability of the configuration knowledge

Stability of the configuration knowledge is quantified by measuring the modifications (changes or insertions) made to the configuration knowledge. PLUSS and Model Templates do not provide specific and separate models to represent the configuration knowledge. As a consequence, this knowledge is scattered throughout the specifications in the form of feature annotations for steps (PLUSS) and

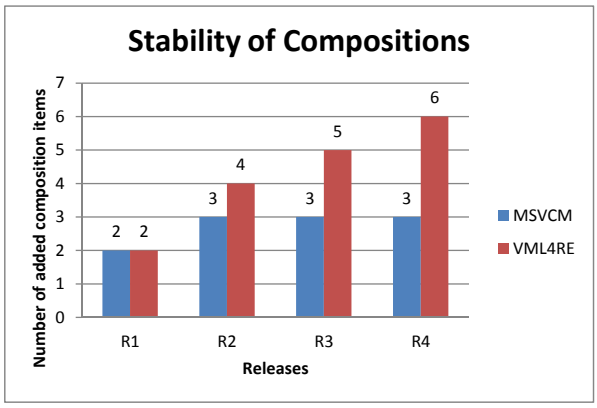
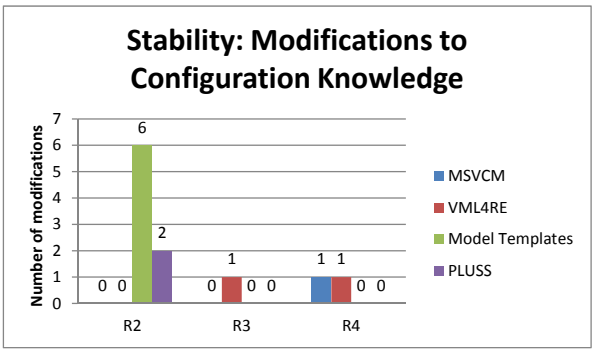
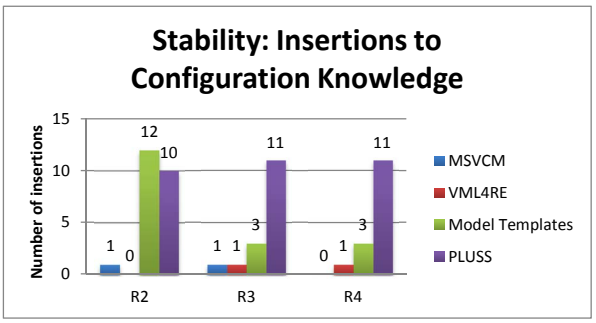


Figure 15: Stability of compositions.



(a) Modifications.



(b) Insertions.

Figure 16: Stability of the configuration knowledge.

stereotypes (Model Templates). While in VML4RE we observe changes and additions of *variants* in the VML4RE composition specification, in MSVCM we observe changes and additions of *configuration items* in the configuration knowledge.

Figure 16 summarizes our findings (lower is better). We can notice that the composition-based approaches tend to be more stable than annotation-based approaches PLUSS and Model Templates. This may be explained by the fact that MSVCM and VML4RE have specific models to deal with the configuration knowledge, while in PLUSS and Model Templates, the knowledge is spread across multiple artifacts. For Model Templates, due to the modularization of the Log Service feature in one diagram, which is reffered by other scenarios, the values for insertions in R3 and R4 (Figure 16(b)) do not grow as much as PLUSS that does not modularize the Log behavior. MSVCM and VML4RE requires the least numbers of insertions and modifications during the evolution scenarios as it will be shown in the Impact metric in Figure 16(b)). Further studies could indicate if that holds true for most cases, and in which cases these compositional techniques would not be of good use.

9. Summary of Results

Figure 17 complements the analysis already provided in Sections 6 to 8 with a summary of the evaluation results for the four CCCMS releases. For each metric we assigned a symbol that helps to distinguish which techniques have good, average or bad results in comparison with the others. The upwards arrow means "good", the rightwards arrow means "average", and the downwards arrow means "bad". The assignment of each symbol was determined automatically using the conditional formatting feature of MS Excel which assigned symbols to series of values based on percentages. In our metrics we used the following percentages limits: bad ≥ 67, average < 67 and ≥ 33, good < 33. For stability and expressivity of the compositions that only applied to two techniques, we used short arrows indicating which one had a better value (upwards arrow) than the other (downwards arrow).

In general, the lower the value obtained for a metric, the better the approach for the correspondent attribute. However, metrics such as Degree of Focus (DoF) and Reachability follow an inverse logic, therefore higher values are interpreted as desired values. For example, in DoS the percentage limits to assign the symbols were: good ≥ 67, average < 67 and ≥ 33, and bad < 33.

Modularity for each technique was measured as the means of DoS and DoF for the four releases. The compositional approaches VML4RE and MSVCM had better results in both DoS and DoF. Their scenario specifications better applied a modular design. VML4RE and MSVCM help

Attribute	Metric	PLUSS	MT	MSVCM	VML4RE
Modulartiy	Average DOS	0,113	0,100	0,058	0,000
	Average DOF	2,960	3,310	4,000	4,000
Stability of specifications	Added steps	32	25	21	21
	Impact	4	5	3	3
Stability of composition	Added compositions	-	-	2,75	4,25
Stability of CK	Modifications	2	6	1	2
	Insertions	32	18	2	2
Expresiv. of compositions	Reachability	-	-	1,5	1,0
Expresiveness of CK	Percentual Growth	321,43	109,62	34,21	134,43

Figure 17: Summary of evaluation results.

to specify each feature separately in only one or few scenarios, which leads to DoS values very close to 0 for MSVCM and 0 for VML4RE. Similarly, VML4RE and MSVCM specified each scenario focusing on only one or few features which resulted in a good DOF=4 in comparison with Model Templates (DOF=3,31) and PLUSS (DOF=2,96). We believe that the annotations mechanism used by Model Templates and PLUSS fail to improve modularity of scenario specifications, even with few features that are scattered through the system such as LOG and Authorization.

Stability of specifications, composition and configuration knowledge (CK) was measured as the sum of all the individual values for stability metrics obtained in all the releases. Similarly, the Impact was measured as the sum of added and modified scenarios in all the releases. The compositional approaches VML4RE and MSVCM obtained the same values for the metrics of stability of the specifications, this means that the most noticeable differences between the compositional approaches (apart from their notation) are found in their expressivity and not in the specification of the scenarios or modularity itself. On the other hand, PLUSS was the best technique to keep almost intact CK specifications (Modifications=2) however, it was done at the price of many insertions (Insertions=32). A different phenomena happens with the rest of the approaches that faced evolution of CK combining few modifications and insertions of CK.

Expressiveness of CK was taken directly from Figure 11. It was measured as the mean of Reachability for the four releases. VML4RE had a low Reachability (=1) compared to MSCVM (=1.5). The Percentual Growth of Expressiveness of CK in MSCVM (=34,21) was the best while in PLUSS it was the worse (=321,43). The results of expressivity of VML4RE are similar to the ones of Model Templates (Percentual Growth of Expressiveness of CK=134.43 and CK=109.62) that does not have any separate configuration knowledge model. We see that the lack or presence of quantification mechanisms affected Expressivity, and Expressivity affected Stability of compositions and CK. For example, the lack of quantification mechanisms in VML4RE limited the Reachability of its pointcuts and influenced negatively the Stability of compositions and CK because of new required variants and compositions items to match elements introduced in new scenarios.

10. Threats to conclusion validity

The type and size of the investigated releases limit our conclusions. However, we chose the CCCMS SPL because the original specifications [14, 29] as well as the the scenarios for each technique[6] are publicly available which helps other researchers to replicate and extend our study.

We believe that CCCMS SPL is a good choice for conducting our assessments. The CCCMS became a benchmarking for SPL development since a call for a special issue of a journal invited for contributions using the same set of requirements detailed here [14, 29]. Therefore, a body of knowledge was established using the CCCMS SPL. In addition, it presents several *Mandatory*, *Optional*, and *Or features*, and some of the optional features change the base specification in a single place, whereas others (e.g., the Log feature) change the specification throughout different scenarios.

With respect to the type and size of the releases, we mainly concentrate our study on increments to the base specification (the first release). Other types of changes were not covered here, such as bug fixes.

[6] http://www.mauricioalferez.com/REJ/REJ-Data.htm

Besides that, some of our conclusions are still valid and could be generalized. For instance, evolving a localized feature specification should not reveal significant differences among the investigated techniques. Differently, if we had to evolve the Log specification, which is not well localized in PLUSS and Model Templates, our assessment procedure would reveal that these techniques are less stable than MSVCM and VML4RE— since several places of the specifications written in PLUSS and Model Templates are likely to change. Moreover, we do not have to change or introduce new requirements to the original SPL specifications, using the releases presented in this paper. Consequently, they have not been proposed to favor any particular approach.

In addition, the chosen metrics suite could also be considered a threat to the validity of our work because it could be engineered to favour one approach over the others. However, some of the metrics (DoS, DoF, reachability, and stability of the compositions and specifications) have been previously used in related works [25, 24]. The other metrics proposed to evaluate the configuration knowledge (expressiveness and stability) are contributions of this paper, and we acknowledge their validation as a matter of future work. Particularly, we use the number of tokens to count the expressiveness of the configuration knowledge. Therefore, our findings regarding this metric depend on the concrete syntax of the current languages used to specify the configuration knowledge in each technique. As a future work, we want to investigate the use of other abstractions to quantify the expressiveness of different representations of configuration knowledge.

11. Related work

There are several works that are connected to our research. Here we introduce these works and relate them to our study.

11.1. Aspect-oriented assessment

Metrics for quantifying scattering and tangling [23, 7] have been applied for assessing modularity in AO programs. In fact, we could have adapted absolute measures for quantifying scattering and tangling in our context, such as Concern Diffusion over Components and Concern Diffusion over Lines of Code [30]. However, absolute measures just reveal if a feature is scattered or not— without any information about the degree of its scattering. In fact, this limitation hinders the comparison of modularity between different specifications. Consequently, we customized a metrics suite proposed by Eaddy and his colleagues [23].

To improve our confidence in the results, we also measure the stability of the specifications and compositions by means of a suite of metrics that had already been validated and used in a previous work [24]. However, here we use these metrics to evaluate stability of aspect-oriented approaches for specifying SPL scenarios, whereas Chitchyan et al. proposed and used those metrics to assess stability of semantic and syntactic aspect-oriented approaches for requirements engineering (AORE). For this reason, we had to extend their metrics suite to assess not only the stability of specifications and compositions, but also the stability and expressiveness of the configuration knowledge.

Previously, some authors of this paper presented a comparison of modularity involving MSVCM and PLUSS [25]. However, here we contribute with a deeper evaluation, considering other quality attributes (such as stability and expressiveness) and additional techniques (Model Templates and VML4RE). Finally, also regarding assessment of aspect-oriented approaches for the earlier stages of software development, Sampaio et al. compared different AORE approaches with respect to the accuracy of resulting specifications and the effort required to build *aspect compositions* [31]. We postpone a similar evaluation in our context to a future work.

11.2. Aspect-oriented requirements engineering

Here we mainly evaluated four approaches for specifying variability in SPL scenarios. Indeed, two of these approaches (MSVCM and VML4RE) are built upon aspect-oriented constructs. Certainly, there are a number of other aspect-oriented approaches for requirements engineering (AORE), such as [32, 33, 34]. Although these works have not been proposed to specify SPL requirements, their support for composition might address the types of variability shown in this paper.

There are other works specifically proposed to represent variability in requirements models. For instance, PLUC [17] extends the use case notation for SPL engineering. It follows an annotative style, and do not present a good separation between the problem and the solution spaces. In fact, we had already investigated PLUC in a previous work [35],

which led us to conclude that PLUC is not maintainable at all.

The model composition language MATA (Modeling Aspects using a Transformation Approach) [36] allows engineers to specify compositions of UML class diagrams, sequence diagrams, and statecharts. Therefore, similarly to MSVCM and VML4RE, MATA also separates the base specification (named Kernel models in [36]) and the variant specifications. In [37] there is a possible adaptation of MATA for modeling scenarios employing sequence diagrams for non-SPL systems.

12. Final Remarks

In this paper, we presented an empirical study that compares and analyzes four existing approaches that address the modeling and management of variabilities in SPL requirements specifications. The investigated approaches are representative of a set of new variability management approaches. They reflect different perspectives of existing approaches, such as: (i) graphics-based versus textual-based; and (ii) annotation versus composition-based. In our study, they were evaluated from the perspectives of modularity, expressiveness and stability through the specification and evolution of a Car Crash Crisis Management System SPL.

Our findings have shown that the composition-based approaches have greater potential to produce stabler and modular SPL requirements specifications. In our study, MSVCM and VML4RE promoted the modular specification of the scenarios and configuration knowledge, which brought more stability to SPL requirements specifications during their respective evolution. It was also observed that the aspect-oriented mechanisms of MSVCM great contributions to improve the modularization of variabilities in the scenario specification. In particular, scenario advices were used to modularize variabilities which are composed with the SPL core scenarios. Also, the absence of quantification mechanisms in the VML4RE specifications have negatively contributed to reduce their expressiveness and stability.

The following recommendations were derived from the analysis and results of our study, which might contribute to the definition of new variability management approaches for requirements specifications:

[(a)]

1. provide support to the separated and modular specification of the configuration knowledge between variability and requirements models;
2. adoption of quantification mechanisms in the specification of the configuration knowledge, aiming at simplifying and improving its expressiveness;
3. use of early aspects techniques to modularize crosscutting scenarios and promote their seamless composition with SPL core requirements thus contributing to variability management.

All these derived guidelines reflect the benefits and advantages that aspect-oriented techniques can bring to the variability management in the SPL requirements specifications. In fact, some of these recommendations are being used to improve MSVCM and VML4RE.

By the metrics we chose, aspect-oriented approaches are superior, particularly where highly cross-cutting features exist, but there may be other unexamined factors, such as learning curve and usability, where the opposite may be true. As a future work, we aim to investigate other quality attributes such as effort and usability of the techniques.

Acknowledgments

This work has been partially supported by (a) the Fundacão para a Ciência e a Tecnologia, Portugal, grant SFRH/BD/46194/2008; (b) the National Program of Academic Cooperation, funded by CAPES, grant Procad Nr 01/ 2007; (c) the National Institute of Science and Technology for Software Engineering (INES), funded by CNPq and FACEPE, grants 573964/2008-4 and APQ-1037-1.03/08; and (d) the EC FP7 STREP project AMPLE: Aspect-Oriented Model-Driven Product Line Engineering.

References

[1] K. Pohl, G. Böckle, F. J. v. d. Linden, Software Product Line Engineering: Foundations, Principles and Techniques, Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2005.

[2] P. Clements, L. M. Northrop, Software Product Lines: Practices and Patterns, Professional, Addison-Wesley, 2001.

[3] K. Czarnecki, U. Eisenecker, Generative programming: methods, tools, and applications, ACM Press/Addison-Wesley Publishing Co. New York, NY, USA, 2000.

[4] L. Bergmans, M. Aksit, Composing crosscutting concerns using composition filters, Commun. ACM 44 (2001) 51–57.

[5] I. F. Alexander, N. Maiden (Eds.), Scenarios, Stories, Use Cases: Through the Systems Development Life-Cycle, John Wiley & Sons, Ltd., 2004.

[6] M. Eriksson, J. Borstler, K. Borg, The pluss approach, domain modeling with features, use cases and use case realizations, in: 9th International Conference on Software Product Lines, LNCS, 2005, pp. 33–44.

[7] E. Figueiredo, et al., Evolving software product lines with aspects: an empirical study on design stability, in: ICSE 2008, ACM, Leipzig, Germany, 2008, pp. 261–270. doi:http://doi.acm.org/10.1145/1368088.1368124.

[8] AMPLE, Ample project, `http://ample.holos.pt/` (2009).

[9] R. Filman, T. Elrad, S. Clarke, M. Ak?it, Aspect-oriented software development, Addison-Wesley Professional, 2004.

[10] AOSD-Association, Aspect-oriented software development community & conference :: Aosd, `http://aosd.net/` (2010).

[11] R. Bonifácio, P. Borba, Modeling scenario variability as crosscutting mechanisms, in: AOSD '09: Proceedings of the 8th ACM international conference on Aspect-oriented software development, ACM, New York, NY, USA, 2009, pp. 125–136. doi:http://doi.acm.org/10.1145/1509239.1509258.

[12] M. Alférez, et al., Multi-view composition language for software product line requirements, in: SLE'09: Proceedings of the 2nd International Conference on Software Language Engineering, 2009.

[13] C. Kästner, S. Apel, M. Kuhlemann, Granularity in software product lines, in: 30th International Conference on Software Engineering, ACM, New York, NY, USA, 2008, pp. 311–320. doi:http://doi.acm.org/10.1145/1368088.1368131.

[14] J. K. N. G. S. Mustafiz, Crisis management systems - a case study for aspect-oriented modeling, Tech. rep., School of Computer Science, McGill University, Montreal, Canada, `http://www.cs.mcgill.ca/ joerg/taosd/TAOSD/TAOSD.html` (June 2009).

[15] M. Eriksson, J. Börstler, K. Borg, Managing requirements specifications for product lines - an approach and industry case study, Journal of Systems and Software 82 (3) (2009) 435 – 447. doi:DOI: 10.1016/j.jss.2008.07.046.

[16] S. Katz, M. Mezini, J. Kienzle (Eds.), Transactions on Aspect-Oriented Software Development VII - A Common Case Study for Aspect-Oriented Modeling, Vol. 6210 of Lecture Notes in Computer Science, Springer, 2010.

[17] A. Bertolino, S. Gnesi, Use case-based testing of product lines, in: ESEC/FSE' 2003, Helsinki, Finland, 2003, pp. 355–358. doi:http://doi.acm.org/10.1145/940071.940120.

[18] H. Gomaa, Designing Software Product Lines with UML: From Use Cases to Pattern-Based Software Architectures, Addison Wesley Longman Publishing Co., Inc., Redwood City, CA, USA, 2004.

[19] K. Czarnecki, M. Antkiewicz, Mapping features to models: A template approach based on superimposed variants, in: R. Glück, M. R. Lowry (Eds.), Generative Programming and Component Engineering, 4th International Conference, GPCE 2005, Tallinn, Estonia, September 29 - October 1, 2005, Proceedings, Vol. 3676 of Lecture Notes in Computer Science, Springer, 2005, pp. 422–437.

[20] S. Zschaler, et al., VML* - a family of languages for variability management in software product lines, in: SLE'09: Proceedings of the 2nd International Conference on Software Language Engineering, 2009.

[21] F. Bachmann, L. Bass, Managing variability in software architectures, SIGSOFT Softw. Eng. Notes 26 (3) (2001) 126–132.

[22] K. Czarnecki, K. Pietroszek, Verifying feature-based model templates against well-formedness ocl constraints, in: GPCE '06: Proceedings of the 5th international conference on Generative programming and component engineering, ACM, New York, NY, USA, 2006, pp. 211–220. doi:http://doi.acm.org/10.1145/1173706.1173738.

[23] M. Eaddy, A. Aho, G. C. Murphy, Identifying, assigning, and quantifying crosscutting concerns, in: ACoM '07: Proceedings of the First International Workshop on Assessment of Contemporary Modularization Techniques, IEEE Computer Society, Washington, DC, USA, 2007, p. 2.

[24] R. Chitchyan, et al., Semantic vs. syntactic compositions in aspect-oriented requirements engineering: an empirical study, in: K. J. Sullivan (Ed.), AOSD 2009, ACM, Charlottesville, Virginia, USA, 2009, pp. 149–160.
URL `http://doi.acm.org/10.1145/1509239.1509260`

[25] R. Bonifácio, P. Borba, Modeling scenario variability as crosscutting mechanisms, in: AOSD 2009, ACM, Charlottesville, Virginia, USA, 2009, pp. 125–136. doi:http://doi.acm.org/10.1145/1509239.1509258.

[26] U. Kulesza, C. Sant'Anna, A. Garcia, R. Coelho, A. von Staa, C. J. P. de Lucena, Quantifying the effects of aspect-oriented programming: A maintenance study, in: ICSM, 2006, pp. 223–233.

[27] M. Lippert, C. V. Lopes, A study on exception detecton and handling using aspect-oriented programming, in: ICSE, 2000, pp. 418–427.

[28] M. Alférez, U. Kulesza, N. Weston, J. Araújo, V. Amaral, A. Moreira, A. Rashid, M. C. Jaeger, A metamodel for aspectual requirements modelling and composition, TechReport D 1.3, Ample-project (2008).

[29] J. Kienzle, N. Guelfi, S. Mustafiz, Crisis management systems: A case study for aspect-oriented modeling, in: T. Aspect-Oriented Software Development [16], pp. 1–22 (2010) 1–22.

[30] E. Figueiredo, et al., On the maintainability of aspect-oriented software: A concern-oriented measurement framework, 12th European Conference on Software Maintenance and Reengineering, 2008. CSMR 2008. (2008) 183–192.

[31] A. Sampaio, et al., A comparative study of aspect-oriented requirements engineering approaches, in: Empirical Software Engineering and Measurement, 2007. ESEM 2007. First International Symposium on, 2007, pp. 166–175. doi:10.1109/ESEM.2007.15.

[32] A. Moreira, A. Rashid, J. ao Araújo, Multi-dimensional separation of concerns in requirements engineering, Requirements Engineering, IEEE International Conference on 0 (2005) 285–296. doi:http://doi.ieeecomputersociety.org/10.1109/RE.2005.46.

[33] R. Chitchyan, A. Rashid, P. Rayson, R. Waters, Semantics-based composition for aspect-oriented requirements engineering, in: AOSD 2007, ACM, Vancouver, British Columbia, Canada, 2007, pp. 36–48.

doi:http://doi.acm.org/10.1145/1218563.1218569.

[34] J. Whittle, et al., An expressive aspect composition language for uml state diagrams, in: International Conference on Model Driven Engineering, Languages and Systems (MODELS'2007), Vol. 4735 of LNCS, Springer, 2007, pp. 514–528.

[35] R. Bonifácio, P. Borba, S. Soares, On the benefits of variability management as crosscutting, in: Early Aspects Workshop at AOSD, Brussels, Belgium, 2008.

[36] P. Jayaraman, et al., Model composition in product lines and feature interaction detection using critical pair analysis, in: G. Engels, B. Opdyke, D. C. Schmidt, F. Weil (Eds.), Model Driven Engineering Languages and Systems, 10th International Conference, MoDELS 2007, Nashville, USA, September 30 - October 5, 2007, Proceedings, Vol. 4735 of Lecture Notes in Computer Science, Springer, 2007, pp. 151–165.

[37] A. Moreira, J. Araújo, The need for early aspects, in: GTTSE, 2009, pp. 386–407.

(a) MSVCM

Feature/Scenario	SC01	SC02	SC03	SC04	SC06	SC07	SC10	ADV01	ADV02
CCCMS	8	-	-	-	-	-	-	-	-
Witness	-	4	-	-	-	-	-	-	-
Int.Resources	-	-	3	-	-	-	-	-	-
Authentication	-	-	-	-	-	-	2	-	2
Ext.Resources	-	-	-	2	-	-	-	-	-
Observe	-	-	-	-	5	-	-	-	-
Rescue	-	-	-	-	-	4	-	-	-
Med.Services	-	-	-	-	-	-	-	2	-

(b) VML4RE

Feature/Scenario	SC01	SC02	SC03	SC04	SC06	SC07	SC10	ADV01
CCCMS	14	-	-	-	-	-	-	-
Witness	-	8	-	-	-	-	-	-
Int.Resources	-	-	4	-	-	-	-	-
Authentication	-	-	-	-	-	-	3	-
Ext.Resources	-	-	-	3	-	-	-	-
Observe	-	-	-	-	8	-	-	-
Rescue	-	-	-	-	-	5	-	-
Med.Services	-	-	-	-	-	-	-	4

(c) PLUSS

Feature/Scenario	SC01	SC02	SC03	SC04	SC06	SC07	SC10
CCCMS	8	-	-	-	-	-	-
Witness	-	4	-	-	-	-	-
Int.Resources	-	-	3	-	-	-	-
Authentication	-	-	2	-	-	-	2
Ext.Resources	-	-	-	2	-	-	-
Observe	-	-	-	-	5	-	-
Rescue	-	-	-	-	-	4	-
Med.Services	-	-	-	-	-	2	-

(d) Model Templates

Feature/Scenario	SC01	SC02	SC03	SC04	SC06	SC07	SC10
CCCMS	14	-	-	-	-	-	-
Witness	-	8	-	-	-	-	-
Int.Resources	-	-	4	-	-	-	-
Authentication	-	-	2	-	-	-	3
Ext.Resources	-	-	-	3	-	-	-
Observe	-	-	-	-	8	-	-
Rescue	-	-	-	-	-	5	-
Med.Services	-	-	-	-	-	4	-

Table 7: Assignment of features to the scenarios' steps in the first release.

(a) MSVCM

Feature/Scenario	SC01	SC02	SC03	SC04	SC06	SC07	SC08	SC09	SC10	ADV01	ADV02	ADV03
CCCMS	8	-	-	-	-	-	-	-	-	-	-	-
Witness	-	4	-	-	-	-	-	-	-	-	-	-
Int.Resources	-	-	3	-	-	-	-	-	-	-	-	-
Authentication	-	-	-	-	-	-	-	-	2	-	2	-
Ext.Resources	-	-	-	2	-	-	-	-	-	-	-	-
Observe	-	-	-	-	5	-	-	-	-	-	-	-
Rescue	-	-	-	-	-	4	-	-	-	-	-	-
Med.Services	-	-	-	-	-	-	-	-	-	2	-	-
Log	-	-	-	-	-	-	-	-	-	-	-	5
Obstacle	-	-	-	-	-	-	-	6	-	-	-	-
Helicopter	-	-	-	-	-	-	6	-	-	-	-	-

(b) VML4RE

Feature/Scenario	SC01	SC02	SC03	SC04	SC06	SC07	SC08	SC09	SC10	ADV01	ADV03
CCCMS	14	-	-	-	-	-	-	-	-	-	-
Witness	-	8	-	-	-	-	-	-	-	-	-
Int.Resources	-	-	4	-	-	-	-	-	-	-	-
Authentication	-	-	-	-	-	-	-	-	4	-	-
Ext.Resources	-	-	-	3	-	-	-	-	-	-	-
Observe	-	-	-	-	8	-	-	-	-	-	-
Rescue	-	-	-	-	-	5	-	-	-	-	-
Med.Services	-	-	-	-	-	-	-	-	-	4	-
Log	-	-	-	-	-	-	-	-	-	-	5
Obstacle	-	-	-	-	-	-	-	8	-	-	-
Helicopter	-	-	-	-	-	-	8	-	-	-	-

Table 8: Assignment of features to the scenarios' steps in the fourth release.

(a) PLUSS

Feature/Scenario	SC01	SC02	SC03	SC04	SC06	SC07	SC08	SC09	SC10
CCCMS	8	-	-	-	-	-	-	-	-
Witness	-	4	-	-	-	-	-	-	-
Int.Resources	-	-	3	-	-	-	-	-	-
Authentication	-	-	2	-	-	-	-	-	2
Ext.Resources	-	-	-	4	-	-	-	-	-
Observe	-	-	-	-	5	-	-	-	-
Rescue	-	-	-	-	-	4	-	-	-
Med.Services	-	-	-	-	-	2	-	-	-
Log	-	-	-	-	5	5	5	5	-
Obstacle	-	-	-	-	-	-	-	6	-
Helicopter	-	-	-	-	-	-	6	-	-

(b) Model Templates

Feature/Scenario	SC01	SC02	SC03	SC04	SC06	SC07	SC08	SC09	SC10	SCLog
CCCMS	14	-	-	-	-	-	-	-	-	-
Witness	-	8	-	-	-	-	-	-	-	-
Int.Resources	-	-	4	-	-	-	-	-	-	-
Authentication	-	-	2	-	-	-	-	-	3	-
Ext.Resources	-	-	-	3	-	-	-	-	-	-
Observe	-	-	-	-	8	-	-	-	-	-
Rescue	-	-	-	-	-	5	-	-	-	-
Med.Services	-	-	-	-	-	4	-	-	-	-
Log	-	-	-	-	1	1	1	1	-	5
Obstacle	-	-	-	-	-	-	-	8	-	-
Helicopter	-	-	-	-	-	-	8	-	-	-

Table 9: Assignment of features to the specification's steps in the fourth release (cont.).

13

Supporting Consistency Checking between Features and Software Product Line Use Scenarios

Authors: Mauricio Alférez, Roberto E. Lopez-Herrejon, Ana Moreira, Vasco Amaral, Alexander Egyed.

Paper Summary: This paper presents the first version of VCC, whose driving objective is to enable consistency checking in the problem space between requirements models such as use scenarios and features. It transforms generic constraints expressions between single features to rules specifically tailored for use scenarios and set of features. Then, it employs propositional formulas to relate these specialized rules to the models involved in the creation of customized use scenarios for specific products. These propositional formulas are produced based on the relationships between: (1) domain constraints that can be obtained from the SPL feature model, (2) the meaning of the relationships between fragments in the use scenarios and SPL features, and (3) a composition model that specifies how to vary SPL use scenarios. Checking if all the products in an SPL satisfy consistency constraints is based on searching for a satisfying assignment of a propositional formula. Therefore, our tool translates propositional formulas that can be evaluated by satisfiability (SAT) solvers. This paper supports part of the consistency checking activity of DCC4SPL (Section 3.5 - Inside VCC) and is exemplified using the VML4RE composition language described in Section 3.4 - Inside VML4RE. However, the version of VCC presented here evolved to the one presented in Chapter 14 - Ensuring Consistency Between Feature Models and Model-Based Specications - The VCC Approach.

Authors Contribution: Mauricio Alférez was the main author and responsible for the main part of the research and writing of this paper, accounting for the 90% of the work. This work started during a staying in the Institute for Software Engineering and Automation at the Johannes Kepler University, Austria where Mauricio worked in close collaboration with Roberto E. Lopez-Herrejon and Alexander Egyed, which are experts in the subject of verification of models. Ana Moreira and Vasco Amaral gave interesting comments mainly in the areas of requirements engineering and model-driven development that helped to improve the content of the paper.

Publication: Published in the proceedings of the 12th International Conference on Software Reuse (ICSR 2011), Pohang, South Korea, June 13-17, 2011 [9, 82]. Acceptance rate: 40%. Conference classification: CORE A.

Supporting Consistency Checking between Features and Software Product Line Use Scenarios

Mauricio Alférez[1], Roberto E. Lopez-Herrejon[2], Ana Moreira[1], Vasco Amaral[1], Alexander Egyed[2]

[1]CITI/Departamento de Informática, Faculdade de Ciências e Tecnologia
Universidade Nova de Lisboa, Caparica, Portugal
[2]Institute for Systems Engineering and Automation
Johannes Kepler University Linz, Austria
{mauricio.alferez,amm,vasco.amaral}@di.fct.unl.pt
{roberto.lopez,alexander.egyed}@jku.at

Abstract. A key aspect for effective variability modeling of Software Product Lines *(SPL)* is to harmonize the need to achieve separation of concerns with the need to satisfy consistency of requirements and constraints. Techniques for variability modeling such as feature models used together with use scenarios help to achieve separation of stakeholders' concerns but ensuring their joint consistency is largely unsupported. Therefore, inconsistent assumptions about system's expected use scenarios and the way in which they vary according to the presence or absence of features reduce the models usefulness and possibly renders invalid SPL systems. In this paper we propose an approach to check consistency —the verification of semantic relationships among the models— between features and use scenarios that realize them. The novelty of this approach is that it is specially tailored for the SPL domain and considers complex composition situations where the customization of use scenarios for specific products depends on the presence or absence of sets of features. We illustrate our approach and supporting tools using *variant* constructs that specify how the inclusion of sets of *variable features* (that refer to uncommon requirements between products of a SPL) adapt use scenarios related to other features.

1 Introduction

A *Software Product Line* (SPL) can be defined as "a set of software–intensive systems sharing a common, managed set of features that satisfy the specific needs of a particular market segment or mission and that are developed from a common set of core assets in a prescribed way"[7]. In SPLs, requirements are organized by *features* that are useful to express product functionalities concisely [19]. There are *common* features between all the products in the product line (sometimes called *mandatory* features), and there are *variable* features that allow distinguishing between products in a product line. In SPL development the *problem space* focuses on variability modeling and describes the different features available in an SPL and their interdependencies. A common representation to model variability are the *feature models,* where features are realized with correspondent artifacts, for example use scenarios diagrams [8].

To produce particular products from a SPL, feature realizations have to be composed according to a specific selection of features from a feature model usually called *product configuration* (also referred to *feature model configuration*). This process requires a mapping between features from a feature model, and artifacts such as use scenarios that realize them. A *use scenario* is a widely used technique that describes, step by step, how an actor is intending to use a system [14]. A number of different approaches have been proposed to create mappings among features and models [13,8,20]. However, ensuring consistency between feature models and recurring requirements specifications techniques such as use scenario modeling has not been thoroughly researched. In this context, by consistency checking we mean the verification of semantic relationships among features and use scenarios. Inconsistent assumptions about system's expected use scenarios and their variations according to the selection of different features, reduce the models usefulness and possibly renders invalid systems. Therefore, it is essential in SPL to determine whether the variability model and its use scenarios defined in the domain requirements specification enable the derivation of any product requirements specification that contains inconsistent requirements.

When a model-based approach is used to represent use scenarios (e.g., in form of use cases or activity diagrams), consistency goes beyond syntactical or semantic errors of each kind of model in isolation. For example, an actor that is not associated with any use case, a dangling node, a loop without exit conditions in activity diagrams or specific set of features that are both simultaneously (and incorrectly) declared as excluding and depending. It means that we aim at taking into account constraints that are not merely expressed in terms of only one language's metamodel which is generally well supported by UML editors in the case of use cases and activity diagrams (e.g., using OCL or hard-coded restrictions particular of each editor) or feature model editors (e.g., using *domain constraints* expressing features interdependencies, and hard-coded restrictions that constrain the construction of the models to conform to their metamodel). In our work, much of consistency checking difficulty lies on maintaining consistency among several, interrelated models. This can become a time-consuming and error prone task given that the number of ways to compose feature realizations grows exponentially with the possible number of SPL features that can be used in a particular product.

In this paper, we present an approach whose driving objective is to enable consistency checking in the problem space between requirements models such as use scenarios and features. It transforms generic constraints expressions between single features to rules specifically tailored for use scenarios and set of features. Then, it employs propositional formulas to relate these specialized rules to the models involved in the creation of customized use scenarios for specific products. These propositional formulas are produced based on the relationships between: i) *domain constraints* that can be obtained from the SPL feature model, ii) the meaning of the relationships between fragments in the use scenarios and SPL features, and iii) a *composition model* that specifies how to vary SPL use scenarios. Checking if all the products in an SPL satisfy consistency constraints is based on searching for a satisfying assignment of a propositional formula. Therefore, our tool translates propositional formulas that can be evaluated by *satisfiability* (SAT) solvers [1]. In case there are constraints that are not satisfied by the SPL, our tool presents to the developer the particular features and fragments of the use scenarios

involved in the violation of the constraint. In our home automation case study this information was useful to take informed decisions about the modifications and additions of domain constraints, use scenarios and its composition specification. The results of the application of our approach are encoraging because they did not show scalability and performance issues, however, we need more extensive validation of our approach with different case studies.

2 Background and Motivation

To understand consistency between features and use scenarios let us introduce first the models we use: features model, use case/activity diagrams, mapping model between features and use cases/activity models, and a composition specification model. After this, we exemplify inconsistency using these models.

2.1 Models Involved in Consistency Checking

Feature Model. A feature model describes a set of all possible valid product configurations [8]. A configuration specifies a concrete product in terms of its features.

Figure 1-1 shows a sample feature model of part of our running example, the *Smart Home* SPL [18]. Smart Home has four optional features, AUTOMATED WINDOWS(AW), AUTOMATED HEATING (AH), REMOTE HEATING CONTROL (RHC) and INTERNET as a mean to control the heater and other devices remotely. Also, it has a set of common features, such as MANUAL WINDOWS and MANUAL HEATING that will be included in all the target products to be produced using the Smart Home SPL.

Specific product configurations can be defined selecting optional features in the feature model 1-1. Figure 1-2 shows a sample product configuration of the Smart Home SPL called PRODUCT-1 that will be used to illustrate consistency problems between features and use scenarios. PRODUCT-1 has all features except AUTOMATED WINDOWS (AW). Domain constraints in the feature model such as the REQUIRES relationship from RHC to INTERNET, can be added incrementally and in parallel with the creation of use scenarios (discussed below).

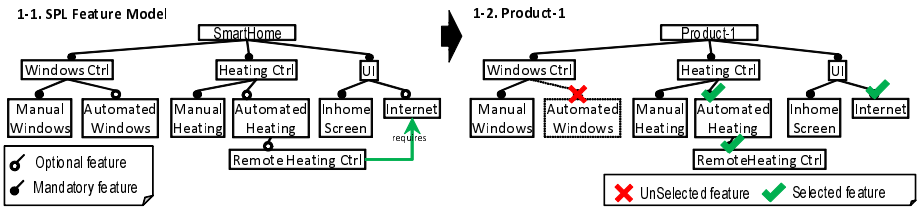


Fig. 1. (1) Simplified sample of the Smart Home feature model; (2) Sample SmartHome configuration that excludes the Automated Windows feature.

Use Scenarios. Features can be realized with other models such as use scenarios. To model use scenarios we employ use case and activity diagrams because they are commonly used in mainstream UML-based methods such as RUP [16] and, in contrast to mere free-form textual scenario descriptions, they help to reduce ambiguity in the specifications [19].

Use case and activity diagrams provide a description of what products in the domain should do. Feature models determine which functionality can be selected when engineering new products from the SPL. Therefore, product requirements specifications consist of customized use cases diagrams and specific paths through those use cases represented in activity diagrams. The customization is guided by a composition specification discussed in next subsection.

Figure 2-1 (Left) shows part of the final target model composed for PRODUCT-1. The INCLUDES relationship describes the case where one use case, the *base* use case, includes the functionality of another use case, the *inclusion* use case. The INCLUDES relationship supports the reuse of functionality in a use case diagram and is used to express that the behavior of the *inclusion* use case is common to two or more use cases. Note that INCLUDES relationships between use cases may constrain the relationship between the features related to them. For example, the INCLUDES relationship between the base use case CTRLTEMPREMOTELY that includes the use case OPENANDCLOSEWINAUTO may imply that feature REMOTEHEATINGCNTRL(SH) requires feature AUTOMATEDWINDOWS (AW). We discuss this and other consistency constraints in Section 3.

Figure 2-1 (Right) shows an activity diagram that depicts the possible scenarios for the use case CNTRLTEMPREMOTELY that comprises activities for the use cases OPENANDCLOSEWINAUTO, CALCENERGYCONSUMPTION and ADJUSTHEATERVALUE. Within this activity diagram it is possible to select several scenarios that correspond to different paths. Two of all the possible scenarios are: Scenario i) includes reaching the in-home temperature and save energy by means of closing some windows, and Scenario ii) to use the heater to reach the desired in-home temperature. It is important to note that the customization of activity diagrams and scenarios depends on the features chosen for the SPL product and also on the relationship with the use case model. For example, in PRODUCT-1 the feature AUTOMATEDWINDOWS was not selected, therefore the WINACTUATOR actor in the use case diagram as well as the swimlane (also called activity partition) related to WINDOWSACTUATOR should not appear in any diagram. Therefore, scenarios such as i) are not realizable because of the lack of windows actuators. This and other constraints will be discussed in Section 3.

Composition Specification To evidence consistency problems between features and use scenarios we employ a composition process (also called, derivation process) for use cases and activity diagrams. Languages such as the VML4RE (Variability Modelling Language for Requirements) [20,4] help to specify how use scenarios can be customized.

Figure 3 illustrates a composition specification that guides the specification of the transformation of requirements specifications of products in the SmartHome SPL. VML4RE [20,4] is a textual language that allows associating *actions*, that wrap a set of model transformations for specific requirements models such as use cases and activity

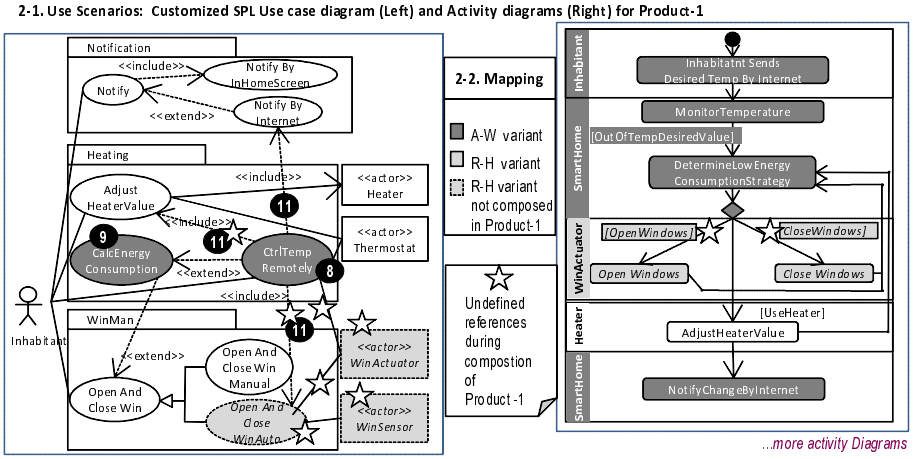


Fig. 2. (1) Referencing undefined model fragments during composition for PRODUCT-1 in the Use Case model (left side) and in the Activity Diagram for the CntrlTempRemotely scenario (right side). (2) Mapping variants to model fragments

diagrams, to combinations of features written as logic expressions that we call *feature expressions*. Feature expressions can be i) *atomic* that represent single features such as "Automated Windows" in Figure 3, Line 1, and ii) *compound* that also contain logic operators such as AND, NOT and OR such as "And ("Remote Heating Ctrl","Automated Heating","Internet")" in line 7. Feature expressions evaluation works as follows: if AUTOMATEDHEATING, REMOTE HEATING CTRL, AUTOMATED HEATING and INTERNET features are selected in a product configuration, the feature expression associated to the variant named "R-H" (i.e., the compound feature expression: And ("Remote Heating Ctrl","Automated Heating","Internet")) will be evaluated to TRUE. The consequence of this is that the actions that are inside the "R-H" variant block (Figure 3, lines 6-13) will be processed and applied to a base model. For example, the CNTRLTEMPREMOTELY use case will be inserted into the package HEATING and then it will be related to other use cases using INCLUDES and EXTENDS relationships. If more than one feature expression is evaluated to TRUE, the default composition order follows a top-down sequence (which corresponds to a left-right sequence in Figure 3).

```
1.  Variant { name : "A-W" for : "Automated Windows"
2.    + UseCase : "OpenAndCloseWinAuto" in Package : "WinMan"
3.    + Partition : "WinActuator" in ActDiagram :"CtrlTempAuto"
4.    ...
5.  }
    ...more variants
```

```
6.  Variant { name : "R-H"
7.    for : And ("Remote Heating Ctrl","Automated Heating","Internet")
8.      + UseCase : "CtrlTempRemotely" in Package : "Heating"
9.      + UseCase : "CalcEnergyConsumption" in Package : "Heating"
10.     Includes from UseCase : "CtrlTempRemotely"  to UseCase(s) :
11.           "NotifyByInternet" and "OpenAndCloseWinAuto" and "AdjustHeaterValue"
12.     ...
13. }
```

Fig. 3. Composition specification of variants A-W and R-H

Mapping Model. Figure 2-1 (Left) and (Right) show use case and activity diagrams fragments, such as actors and use cases, related with the variants shown in Figure 3. The base mechanism to relate requirements model fragments to features is to use a correspondence table (or mapping table), as presented by [11], [19] and [3]. In our case, we parse the composition specification to generate the mapping between variants and parts of the use cases, therefore, for example if variant named A-W inserts the OPENANDCLOSEWINAUTO use case, we link A-W to OPENANDCLOSEWINAUTO. To facilitate the visualization of such relationships with the models, in the figure we assign different gray tones to the models fragments according to the features that they are related to (see mapping in Figure 2-2). Please note that specific model fragments could be related also to more than one variant. This may be considered as a m-to-n (m and n >= 1) mapping between variants and model fragments and is not illustrated in Figure 2.

2.2 Consistency Checking Motivation

Consistency checking has to ensure that inconsistent requirements do not become part of the requirements specifications of a given product. Our work aims at guaranteeing that *all* the products that could be derived from a feature model indeed have consistent requirements specifications. This is achieved through the description and verification of semantic relationships between feature model and use scenarios. One of the possible inconsistencies between features and use scenarios in the Smart Home SPL happens between the relationship of variants R-H and A-W, and the INCLUDES relationship between the use cases CNTRLTEMPREMOTELY and OPENANDCLOSEWINAUTO which are related to R-H and A-W variants respectively. The domain requirements are:

R1- Only one, none or both R-H and A-W variants can be included in a product. (This is implicit in the feature model and composition model because all the features in the feature expression of R-H variant are optional (i.e., REMOTE HEATING CNTRL, AUTOMATED HEATING and INTERNET are optional features), and the only feature in the feature expression A-W is also optional (i.e., the AUTOMATED WINDOWS feature is optional)); and

R2- If the use case CNTRLTEMPREMOTELY is provided in a product then the use case OPENANDCLOSEWINAUTO must be provided too, (This is implicit in the includes relationship from the use case CNTRLTEMPREMOTELY to OPENANDCLOSEWINAUTO in the use case diagram in Figure 2-1 (Left)).

Figure 2-1 shows PRODUCT-1 built using the composition model shown in Figure 3. In PRODUCT-1 the feature expression of variant R-H (3, line 7) evaluates to TRUE. However, because Figure 1-2 does not include the AUTOMATED WINDOWS feature, the feature expression of variant A-W (i.e., AUTOMATED WINDOWS) (3, Line 1) evaluates to FALSE and the actions inside its variant block are not processed. We annotated the diagrams with numbers that represent the line in Figure 3 where a composition action is specified. Note that we omitted some of the actions, for example, the insertion of some actors such as WINSENSOR and WINACTUATOR and some partitions such as HEATER.

PRODUCT-1 presents inconsistent requirements *R1* and *R2*. This is evident during composition of use scenarios. See lines 10-11 when the action “Includes from UseCase

: "CtrlTempRemotely" to UseCase(s) : "NotifyByInternet" and "OpenAndCloseWinAuto" and "AdjustHeaterValue" ” references elements such as the use case OPENANDCLOSEWINAUTO that do not exist in the model. In this case, PRODUCT-1 fulfills requirement *R1*, but not requirement *R2*. The result is that the functionality provided by OPENANDCLOSEWINAUTO will not be present in the requirements of PRODUCT-1 and therefore it will not be taken into account in later stages of its development process.

It is not too difficult to check consistency manually in small examples with a reduced number of features such as the one mentioned previously. One solution to solve the inconsistency for our example would be to guarantee the presence of the feature AUTOMATED WINDOWS when AUTOMATIC HEATING or REMOTE HEATING CTRL are selected, in every possible feature model configuration using a domain constraint REQUIRES. Another solution is to establish that AUTOMATED WINDOWS will be a mandatory feature in the SPL. However, the number of possible feature combinations may grow exponentially with the number of features of the SPL. The result of this explosion is that it becomes unfeasible to manually check the consistency of all the products.

To guarantee that all the products that could be derived from a feature model indeed have consistent requirements specifications we take into account the relationships between domain requirements specified using use scenarios and feature models to propose rules and constraints to support consistency checking in SPLs use scenarios as it is shown in the next section.

3 Consistency Checking between Features and Use Scenarios

While some product configurations of a feature model may generate consistent use scenarios, other product configurations based on the same feature model could lead to inconsistencies in the requirements specifications. In this section we present our approach for consistency checking between SPL features and use scenarios.

3.1 Approach Overview

Figure 4 presents an overview of our approach. Section 2 explained and exemplified the specification of a feature model, use scenarios (Figure 4, Step 1), and the mapping between variants and fragments of the use scenarios (Step 2). Based on previous work [17], we have developed a consistency checking approach for use scenario composition based on variants. This approach relies on the domain evaluation of feature expressions, written as propositional formulas that are associated to a *variant* and transformations of use scenarios called *actions*. We denote D_f the domain constraints that can be derived from a feature model of an SPL and are expressed in terms of atomic features f (Step 3), and C_{VAR_f} denote composition constraints that will be derived in next section (Step 4) and are expressed in terms of variants (VAR_f). We use propositional logic to express and relate D_f and C_{VAR_f}(Step 5). Because we are interested in verifying that all members of the product line satisfy a given composition constraint, Equation 1 should not be “satisfiable”. If it is satisfied, it means that there is a product of the product line that does not meet constraint C_{VAR_f}. The violating product configurations can be identified

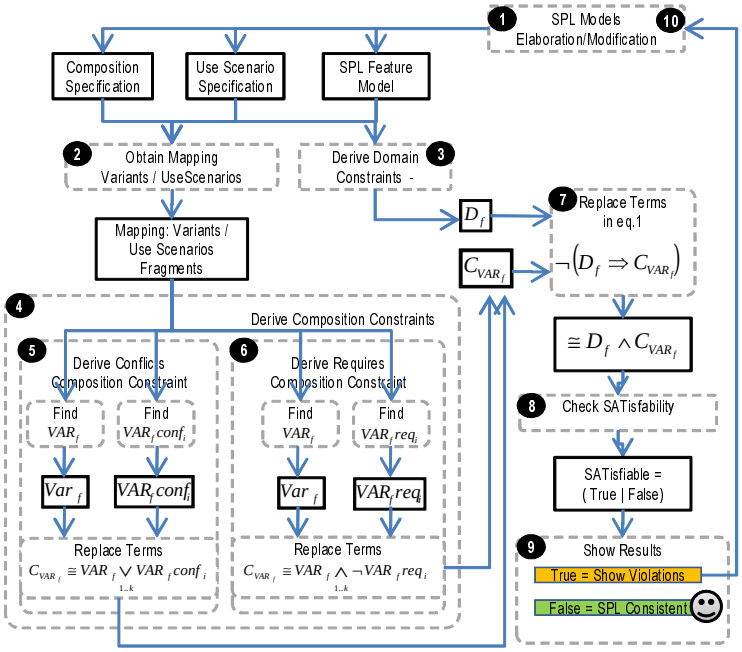


Fig. 4. Overview of Our Consistency Checking Approach

using a SAT solver (Step 7 and 8). This can support the developer to take informed decisions on modifications of the initial SPL models, for example, creating or modifying domain constraints (Step 10).

$$\neg \left(D_f \Rightarrow C_{VAR_f}\right) \tag{1}$$

Section 2-1 shows that at least one product (i.e., PRODUCT-1) from the products that can be configured based on the feature model of the Smart Home SPL is inconsistent. In that case, *composition constraints* (also called *implementation constraints*) between the elements in use scenarios such as the INCLUDES between use cases, imply the application of domain constraints for example, turning the AUTOMATEDWINDOWS feature from optional to mandatory or creating a REQUIRES dependency (also called domain constraint) from AUTOMATEDHEATING to AUTOMATEDWINDOWS. That particular inconsistency that will help to explain our approach can be defined as:

– *Rule Required Inclusion Use Case:* at least one variant $(VAR_f req_i)$, defines an inclusion use case that must be selected in every feature configuration that contains the variant (VAR_f) which introduces a base use case linked to the inclusion use case.

3.2 Deriving Domain Constraints (D_f)

Figure 4 - Step 3 shows that the domain constraints are derived from a SPL feature model. Therefore the D_f in a SPL is the same for all the possible products configura-

tions and do not vary depending on the consistency rule. Using a well-known translation table between feature models and propositional formulas (see Figure 5) helps to get D_f in Equation 1. In Equation 2 we only show the HEATING-CTRL branch because it is the most complex branch in Figure 1-1 and relates directly with our exemplar "Required Inclusion Use Case" rule. The translation obtained in the first line of Equation 1 means that all products unconditionally must contain the root feature SMARTHOME. The second line means that given that HEATING CTRL is a mandatory feature, it must be included in all the products. The third line means that MANUALHEATING is included in all the products that include their parent feature (i.e., HEATINGCTRL), in contrast to AUTOMATEDHEATING and REMOTEHEATINGCTRL (lines 3-4), that may be or not included when their respective parents HEATINGCTRL and AUTOMATEDHEATING are included in a product. Line 5 means that REMOTEHEATINGCTRL *requires* of the INTERNET feature.

$$
\begin{aligned}
&1.(SmartHome \Leftrightarrow TRUE) \wedge \\
&2.\ (SmartHome \Leftrightarrow HeatingCtrl) \wedge \\
&3.\ (HeatingCtrl \Leftrightarrow ManualHeating) \wedge (AutomatedHeating \Rightarrow HeatingCtrl) \wedge \\
&4.\ (RemoteHeatingCtrl \Rightarrow AutomatedHeating) \wedge \\
&5.\ (RemoteHeatingCtrl \Rightarrow Internet)
\end{aligned}
\tag{2}
$$

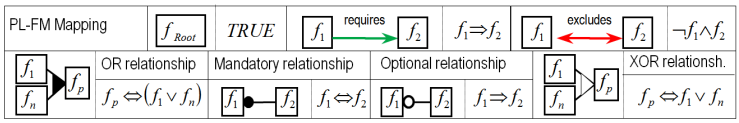


Fig. 5. Mapping from Feature Model to Propositional Logic [6].

In this section we addressed D_f, the first part of Equation 1. Next section presents C_{VAR_f} that comprises a set of constraints that are essential for consistency between use scenarios and the set of domain constraints expressed in Equation 2.

3.3 Deriving Composition Constraints (C_{VAR_f})

Composition constraints act as consistency rules describing the semantic relationships that must hold among the different models. Figure 4-4 shows two kinds of composition constraints that can be expressed in propositional logic. We classified them according to the type of domain constraint that they relate with: i) a constraint that implies a REQUIRES relationship between features that therefore implies dependencies between variants (Figure 4- Step 4), and ii) a constraint that implies a EXCLUDES relationship (Figure 4- Step 6) between features and therefore implies incompatibilities between variants (Figure 4- Step 5). This section shows those constraint equations expressed in propositional logic.

EXCLUDES Relationship*:* Let VAR_f be a variant that defines a model element e. A variant $VAR_f conf_1$ conflicts with VAR_f if $VAR_f conf_1$ defines a model element c which cannot be present in the same requirements specifications of a product where element e is also present. Therefore, because of the incompatibility between elements

e and c, if variant VAR_f is selected then variant $VAR_f conf_{\text{I}}$ should not be selected in the same product configuration. This is denoted in the following expression where k represents the number of variants in the composition specification:

$$C_{VAR_f} \equiv VAR_f \Rightarrow \neg \left(\bigvee_{1..k} (VAR_f conf_{\text{i}}) \right) \equiv \neg VAR_f \vee \neg \left(\bigvee_{1..k} (VAR_f conf_{\text{i}}) \right) \quad (3)$$
$$\equiv VAR_f \wedge \bigvee_{1..k} VAR_f conf_{\text{i}}$$

REQUIRES Relationship*:* Let VAR_f be a variant that refers to a model element e defined by another variant. To be consistent, the requirements specifications of a product that includes variant VAR_f must also include at least one other variant $VARreq_{\text{I}}$ (required variant) where element e is defined. This is denoted in the following expression where k represents the number of variants in the composition model:

$$C_{VAR_f} \equiv VAR_f \Rightarrow \bigvee_{1..k} (VAR_f req_{\text{i}}) \equiv \neg VAR_f \vee \bigvee_{1..k} (VAR_f req_{\text{i}}) \quad (4)$$
$$\equiv VAR_f \wedge \bigwedge_{1..k} \neg VAR_f req_{\text{i}}$$

The rule "Required Inclusion Use Case" mentioned at the beginning of this section is an example of this last kind of constraint expression. An instance of this constraint is found in our motivation example related to the use scenario of CNTRLTEMPREMOTELY. For example, given that the variant VAR_f = R-H is selected (i.e., a product with REMOTE HEATING CNTRL, AUTOMATED HEATING and INTERNET features), and it is related to the base use case CTRLTEMPREMOTELY, we want to guarantee that there are at least one variant (e.g., $VAR_f\ req_{\text{I}}$ = A-W) related to the inclusion use case OPENANDCLOSEWINAUTO (i.e., model element e = use case OPENANDCLOSEWINAUTO), and that its feature expression evaluates to TRUE in all possible feature model configurations. This way, we guarantee the presence of the functionality required by CTRLTEMPREMOTELY, such as to include a WINDOWSACTUATOR that regulates the temperature opening and closing windows. Thus, we can get a constraint instance replacing the variants by their corresponding feature expressions:

$$(\, RemoteHeatingCntrl \wedge AutomatedHeating \wedge Internet \,) \quad (5)$$
$$\wedge \neg (\, AutomatedWindows \,)$$

3.4 Replacing Terms in Equation

The replacing step depicted in Figure 4- Step 7 depends on the kind of constraint that we created in previous section. If we replace C_{VAR_f} of Equation 4 in Equation 1 and perform some logic manipulation to translate expressions of the form $x \Rightarrow y$ to $\neg x \vee y$, and $x \vee y$ to $\neg x \wedge \neg y$ respectively, we obtain the expression in Equation 6.

$$REQUIRES: \neg \left(D_f \Rightarrow \left(VAR_f \wedge \bigwedge_{1..k} \neg VAR_f req_{\text{i}} \right) \right) \equiv D_f \wedge VAR_f \wedge \bigwedge_{1..k} \neg VAR_f req_i \quad (6)$$

Similarly, if we replace C_{VAR_f} of Equation 3 in Equation 1, and perform some logic manipulation, we obtain the expression in Equation 7.

$$EXCLUDES: \neg \left(D_f \Rightarrow \left(VAR_f \wedge \bigvee_{1..k} VAR_f conf_{\text{i}} \right) \right) \equiv D_f \wedge VAR_f \wedge \bigvee_{1..k} VAR_f conf_i \quad (7)$$

3.5 Checking SATisfability

Figure 4- Step 8 shows that the input for satisfability checking are expressions such as the ones in 6 and 7. Each expression to be checked is instantiated with:

i) the specific domain constraints, D_f of the SPL produced in Equation 2,

ii) the feature expressions related to the variants VAR_f and either the set of required variants$VAR_f\ req_i$, or the set of conflictant variants $VAR_f\ conf_i$.

Equation 4 evaluates to true when any action inside variant VAR_f requires an element or set of *required elements* that are not composed in the use scenarios. It happens because none of the correspondent variants $VAR_f\ req_i$ that introduce the *required elements* was selected in the product configuration. Also, expression 3 evaluates to false when variant VAR_f defines an element or set of elements that are introduced in the use scenarios that also contain elements defined by other variant(s) $VAR_f\ conf_i$.

3.6 Show Results and SPL Models Modification

The possible results generated by a SATisfability checker for each expression (Figure 4- Step 9) can be TRUE (satisfiable) or FALSE (insatisfiable). In case we obtain FALSE for all the expressions, we know that the SPL is consistent because there are not inconsistencies between the relationships and dependencies (e.g., excludes, optional, mandatory, requires) between features depicted in the SPL feature model, and the use scenarios. In case we obtain a TRUE in an expression, our tool based on the mapping between variants and model elements in the use scenarios shows a list of the variants and the model fragments related to the inconsistency. Taking the example of the Smart Home feature model depicted in Figure 1, the result of the SAT solver for the Rule - Required Inclusion Use Case is that it is satisfiable (i.e., it evaluates to TRUE). Which means that there is an inconsistency between the features and use scenarios. An example of the type of message generated by our tool to the user 4 is:

> "...Inconsistent use scenario(s) [CTRLTEMPREMOTELLY] and feature(s) in feature expression(s) of variant(s) [A-W], [R-H]. The Action: [Includes from UseCase: "CtrlTempRemotely" to Use Case(s) "OpenAndCloseWinAuto"] implies a [REQUIRES] relationship between variant [R-H] and required variant(s) [A-W] that is not enforced in the SPL feature model...".

Based on this information, for the SAT solver to evaluate to FALSE, the developers may consider for example to:

- Modify the feature model: the set of SPL domain constraints that can be extracted from the feature model can be modified for example creating a REQUIRES relationship for AUTOMATEDHEATING feature to AUTOMATEDWINDOWS, or changing the AUTOMATEDWINDOWS feature from optional to mandatory.

- Modify use scenarios and composition model: for our particular rule, developers may want to check if in fact the INCLUDES association between use cases CTRLTEMPREMOTELY and OPENANDCLOSEWINAUTO is mandatory for every single product or not.

4 Tool Support

Tools for consistency checking can be highly effective for detecting errors in SPL requirements specifications. Such tools not only can find errors people miss, but also they can alleviate developers from the tedious and error-prone task of checking requirements specifications for consistency. Our tool prototype Variability Consistency Checker for Requirements (VCC4RE) [2] was designed to support the process described in Section 3.1 and consist on several components: (i) composition models editor for the VML4RE language, (ii) two translators: one from propositional formulas in prefix notation to conjunctive normal (CNF) form in DIMACS format [1], and the other from the CNF clauses provided by the feature model editor to DIMACS format; and finally (iii) the consistency checker.

We created the *composition model editor* using EMFTEXT [1]. It provides the software infraestructure to derive an initial concrete syntax and plug-in based on the metamodel of our VML4RE language written in Ecore [2]. We employ this technology mostly because of two reasons: first, it separates concrete syntax and abstract syntax which eases the maintenance of the language, and second, it provides a default Human Usable Notation (HUTN) [3] as concrete syntax. Using the HUNT concrete syntax in comparison with our previous tool version [20] allows a more usable and suitable notation for describing requirements composition.

We created a *translator for feature models* created with the SPLOT editor [4]. We chose SPLOT because it allows us to share and edit our models collaboratively via web, and because it generates the CNF formula that represents the domain constraints (D_f) in our equations that later we transform to a widely accepted standard format for boolean formulas in CNF called DIMACS.

Also, we created another *translator* to obtain the feature expressions related to each variant in $VAR_f \wedge \bigwedge_{1..k} \neg VAR_f req_{\mathrm{i}}$ and $VAR_f \wedge \bigvee_{1..k} VAR_f conf_{\mathrm{i}}$ from our composition model. It translates from a prefix notation of propositional formulas of our composition specification, to CNF formulas in DIMACS format. Composition model, consistency rules, as well as the use cases and activity diagrams modelled in any Ecore-based UML tool are interpreted by our consistency checker to produce a set of constraints expressions in CNF DIMACS format. Then, it is possible to use a standard SAT solver to determine the satisfability of each formula. In our case, we experimented with PicoSAT [5] and SAT4J[6].

5 Evaluation

The complete Smart Home SPL was used to evaluate our approach. We chose this case study because, despite of being a large-scale embedded system, this can be understood

[1] http://www.emftext.org/: Concrete syntax mapper
[2] http://www.eclipse.org/modeling/emf/: Eclipse Modelling Framework based on Ecore
[3] http://www.omg.org/spec/HUTN/: The OMG HUTN specification.
[4] http://www.splot-research.org/: Software Product Line Online Tools
[5] http://fmv.jku.at/picosat/: PicoSAT: Pico satisfability solver
[6] http://sat4j.org/: SAT for Java

by a general reader given its application in everyday's life. Also, we had previous experience modelling variability and part of the use scenarios of the Smart Home supported by one of our industrial partners who set the requirements of the system [18].

Features	59	Variants	27
CNF clauses	79	Rules	6
Use Cases	36	Rule instances checked	74
Activity Diagrams	13	Domain constraints created after consistency checking	16
Scenarios	48	Time taken in consistency checking in milliseconds	810

Table 1. Evaluation results using VCC4RE in the Smart Home SPL

Table 1 summarizes some information about the evaluation. The Smart Home has 59 features and comprises significant aspects of modern home automation domain such as security, HVAC (Heating, Ventilating, and Air Conditioning), illumination control, fire control and multiple user interfaces. These features describe variability at the use scenarios therefore, it is relevant to all kind of SPL stakeholders which are not necessarily experts in domotics and its implementation technologies. When mapped to propositional formulas the feature model produced 79 clauses in CNF format.

We modelled the use scenarios manually using an open source Ecore-based UML tool called Papyrus [7]. In total we modelled 36 use cases, 13 activity diagrams that can represent 48 different possible scenarios, and an initial set of 6 rules for use scenario consistency that follow a very similar reasoning than the rule *Required Inclusion Use Case* explained in Section 2. They vary only in the kind of model elements and their relationships with other model elements, for example: inclusion, generalization, specialization, aggregation and mapping between activity diagram partitions to actors and use cases. Based on the scenarios and feature model we specify 27 *variant* modules using VML4RE. Before applying our approach for consistency checking, we found that using the Smart Home feature model it was possible to generate ONE BILLION product configurations. This information can be obtained using the feature model analyzer provided by the SPLOT tool and allows us to evidence the complexity of checking consistency without any approach and tool support such as the one that we proposed in this paper.

In our experiments we found in total 74 rules instances to check. Using this information we created 16 domain constraints, mainly dependencies of type REQUIRES between features in the feature model that finally help us to solve consistency between use scenarios and features. 16 errors is a significant number taking into account mainly two things: i) Use scenarios, feature model and composition were first carefully modeled and before applying our approach they were apparently "perfect", and ii) The large number of possible combinations of features, the number of variants and use scenarios makes this task challenging, however our approach and tool support gives results in a "blink of an eye". The time taken to evaluate consistency rules using the Pico SAT solver and produce the results is in the order of milliseconds when run on an Intel

[7] http://www.eclipse.org/modeling/mdt/?project=papyrus : Papyrus

Core-Duo i5 at 2.4 Ghz. Given that in VCC4RE, feature models and constraints are mapped to clauses, the performance and scalability of our approach are proportional to the efficiency of the SAT solvers which are able to handle large number of clauses in industrial applications. However, though encouraging results, the scalability of our approach needs to be more extensively validated with more complex case studies and probably using more consistency rules. Doing that is part of our future work.

6 Discussion and Related Work

An issue in the development of SPLs is the lack efficient approaches for consistency checking among all the artifacts, including requirements specifications. In model-driven development this becomes a crucial issue as software is built by means of a chain of transformations. This can start from assets such as requirements specification models, to code-based assets that typically depend on a particular implementation technology. In this setting, the quality of the final code of target products depends mostly on (i) the transformations, (ii) the source models of each transformation and (iii) the information added after each transformation. Therefore, to create constraints helps not only to compose models that helps to understand the intended products to the SPL stakeholders, but also to obtain good quality source models that are the base for deriving good quality code.

The idea of this paper was to explore whether it was possible to use so called "hard" methods for consistency checking as early as requirements analysis. Usually such methods are used much later in the development. We believe now that they can be used much earlier and therefore some inconsistencies do not have to be left until later to be detected. The use of these methods is transparent for the SPL developer and therefore, it does not add extra complexity to the modeling process. SAT solvers are implemented by libraries that are used internally by VCC4RE.

The effective use of use scenarios in SPL demands mechanisms for consistency checking that cope with variability. However, to the best of our knowledge, this issue has not been extensively researched except by Czarnecki, et al [9]. They observed that implementation constraints should follow from domain constraints. Their findings apply to a different composition technique that uses model templates to generate concrete models for product configurations. That work ensures that no ill-structured template instances (i.e., concrete models of products) will be generated from a correct product configuration. In comparison with that work, we check consistency between use scenarios and feature models of domain requirements specifications and we do not assume that the feature model contains all domain constraints since its creation as it usually happens in incremental SPL development processes. In fact, our approach benefits from the semantic of the use scenarios to deduce domain constraints.

There are different research areas related to our work and that have been taken into account the importance of consistency constraints in models. In the field of well-formedness of models for example Egyed [10]. Also, for single systems modeling, Jacobson [15] used aspect-oriented use case models. However, none of those works check consistency of SPL models, and their composition mechanism does not support model

weaving of model fragments as it is possible with a requirements-tailored composition language as VML4RE.

Previous work [17] addressed consistency in composition in multi-view modeling in SPL following a FOSD [5] approach for models closer to the product implementation. Also, Harhurin and Hartmann [12] provided denotational semantics and a notation called *Service Diagram* to describe system functionality and variability. Both works focus only on depedencies between atomic features. Our work addresses composition of requirements specifications and an advanced way for model composition based on an aspect-oriented framework VML4RE that is capable to manage variants in addition to atomic features.

7 Conclusions and Future Work

This paper establishes constraints and presents tool support for consistency checking between use scenarios and features in the SPL domain, using feature models and VML4RE. However, our approach does not depend on the use of VML4RE. We use it because its actions facilitate expressing the composition in use scenarios. The objective of checking consistency is to guarantee that all the products that could be derived from a feature model indeed have consistent requirements specifications. This means without omitting information or containing conflicting requirements that eventually may cause errors when transformed and implemented into more platform dependent models and code.

The feasibility of our approach was evaluated using a prototype tool and a home automation case study. The results show that performance and scalability were not an issue. However, these aspects need further assessment with larger and more complex SPLs and consistency rules. Such assessment is part of our future work.

We think that the application of constraints is necessary but do not satisfy completely the problem of consistency checking of models. This problem also depends on the composition order of the variants and in the application order of the actions inside each variant block. Currently, we are researching algorithms to calculate the precedence order between variants and its application in non-monotonic composition. Our proposal here is a proof of concept. Our strategy can be extended for other models, for example to model variability of system qualities, that is not within the scope of our paper and is part of our future work. Here, we are addressing part of the problem for some models.

8 Acknowledgements

This work was partially supported by the CITI, Portugal, the European project AMPLE, contract IST-33710 and the grant SFRH/BD/46194/2008 of Fundação para a Ciência e a Tecnologia, Portugal. It was also partially funded by the Austrian FWF under agreement P21321-N15 and Marie Curie Actions—IEF project number 254965. We thanks to Alexander Nöhrer for its Java interface for PicoSAT.

References

1. Int. confs. on theory and applications of satisfiability testing, http://www.satisfiability.org/
2. Alférez, M.: Variability consistency checking for requirements tool, http://citi.di.fct.unl.pt/prototype/prototype.php?id=116
3. Alférez, M., Kulesza, U., Sousa, A., Santos, J., Moreira, A., Araújo, J., Amaral, V.: A model-driven approach for software product lines requirements engineering. In: SEKE. pp. 779–784 (2008)
4. Alférez, M., Santos, J., Moreira, A., Garcia, A., Kulesza, U., Araújo, J., Amaral, V.: Multi-view composition language for software product line requirements. In: SLE. pp. 103–122 (2009)
5. Batory, D.: Ahead tool suite, hhttp://www.cs.utexas.edu/users/schwartz/ATS.html
6. Benavides, D., Segura, S., Cortés, A.R.: Automated analysis of feature models 20 years later: A literature review. Inf. Syst. 35(6), 615–636 (2010)
7. Clements, P., Northrop, L.: Software Product Lines: Practices and Patterns. Addison-Wesley, Boston, MA, USA (2002)
8. Czarnecki, K., Eisenecker, U.W.: Generative programming: methods, tools, and applications. ACM Press/Addison-Wesley Publishing Co., New York, NY, USA (2000)
9. Czarnecki, K., Pietroszek, K.: Verifying feature-based model templates against well-formedness ocl constraints. In: Proc. of the GPCE'06. pp. 211–220. GPCE '06, ACM, New York, NY, USA (2006)
10. Egyed, A.: Fixing inconsistencies in uml design models. In: Proc. of the 29th Int. Conf. on Software Engineering. pp. 292–301. ICSE '07, IEEE Computer Society, Washington, DC, USA (2007)
11. Gomaa, H.: Designing Software Product Lines with UML: From Use Cases to Pattern-Based Software Architectures. Addison Wesley Longman Publishing Co., Inc., Redwood City, CA, USA (2004)
12. Harhurin, A., Hartmann, J.: Towards consistent specifications of product families. In: FM. pp. 390–405 (2008)
13. Heidenreich, F., Kopcsek, J., Wende, C.: Featuremapper: mapping features to models. In: Companion of the 30th Int. Conf. on Software Engineering. pp. 943–944. ICSE Companion '08, ACM, New York, NY, USA (2008)
14. Jacobson, I.: Object-Oriented Software Engineering: A Use Case Driven Approach. Addison Wesley Longman Publishing Co., Inc., Redwood City, CA, USA (2004)
15. Jacobson, I., Ng, P.W.: Aspect-Oriented Software Development with Use Cases (Addison-Wesley Object Technology Series). Addison-Wesley Professional (2004)
16. Kruchten, P.: The Rational Unified Process: An Introduction. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 3 edn. (2003)
17. Lopez-Herrejon, R.E., Egyed, A.: Detecting inconsistencies in multi-view models with variability. In: ECMFA. pp. 217–232 (2010)
18. Morganho, H., Gomes, e.a.: Requirement specifications for industrial case studies. Deliverable D5.2, Ample Project (2008), www.ample-project.net
19. Pohl, K., Böckle, G., Linden, F.J.v.d.: Software Product Line Engineering: Foundations, Principles and Techniques. Springer-Verlag New York, Inc., Secaucus, NJ, USA (2005)
20. Zschaler, S., Sánchez, P., Santos, J., Alférez, M., Rashid, A., Fuentes, L., Moreira, A., Araújo, J., Kulesza, U.: Vml* - a family of languages for variability management in software product lines. In: SLE. pp. 82–102 (2009)

14

Ensuring Consistency Between Feature Models and Model-Based Specications - The VCC Approach

Authors: Mauricio Alférez, Roberto E. Lopez-Herrejon, Ana Moreira, Vasco Amaral, Alexander Egyed.

Paper Summary: This paper proposes our approach to support consistency checking between a feature model and its corresponding model-based specifications. The resulting approach is called Variability Consistency Checking (VCC). VCC employs propositional formulas to relate constraints between features in feature models and constraints inferred from model-based specifications. Checking if all the products in an SPL satisfy a consistency constraint is based on searching for a satisfying assignment of a propositional formula. VCC and its supporting tool were validated based on two case studies from different application domains. This paper supports the consistency checking activity of DCC4SPL described in Section 3.5 - Inside VCC.

Authors Contribution: Mauricio Alférez was the main author and responsible for the main part of the research and writing of this paper, accounting for the 90% of the work. This work started during a staying in the Institute for Software Engineering and Automation at the Johannes Kepler University, Austria. All the authors gave interesting comments that helped to improve the content of the paper.

Publication: This paper is under revision by an international journal.

Ensuring Consistency between Feature Models and Model-Based Specifications - The VCC Approach

Mauricio Alférez[a,*], Roberto E. Lopez-Herrejon[b], Ana Moreira[a], Vasco Amaral[a], Alexander Egyed[b]

[a] *CITI/Departamento de Informática, Faculdade de Ciências e Tecnologia, Universidade Nova de Lisboa, Caparica, Portugal*
[b] *Institute for Systems Engineering and Automation, Johannes Kepler University, Linz, Austria*

Abstract

Context: Software Product Line Engineering (SPLE) is a successful paradigm to produce a family of product variants for a specific domain. A challenge in SPLE is to ensure that the specification of all products (expressed in a feature model), does not diverge from the specification of what can be produced (based on other model-based specifications). This challenge is difficult to address due to the high number of possible combinations of features and model fragments.

Objective: The objective of this paper is to propose an approach to support consistency checking between a feature model and its corresponding model-based specifications.

Method: The resulting approach is called Variability Consistency Checking (VCC). VCC employs propositional formulas to relate constraints between features in feature models and constraints inferred from model-based specifications. Checking if all the products in an SPL satisfy a consistency constraint is based on searching for a satisfying assignment of a propositional formula. VCC and its supporting tool were validated based on two case studies from different application domains.

Results: The results related to consistency between models demonstrate the value of VCC from four different dimensions: (1) VCC can be applied to early model-based SPL development such as requirements and architecture specifications, (2) VCC considers complex composition situations where the customization of models for specific products depends on the presence or absence of sets of features, (3) VCC checks consistency of all possible products at once and not only for a small set of them, and (4) since our focus is on early development, VCC does not assume a correct and complete feature model; instead, it helps developers to complete the feature model based on the detected inconsistencies with respect to other models.

Conclusion: VCC guarantees consistency between a feature model and other models that specify its features without forcing the composition of the models for all the possible product variants. We argue that VCC is useful in early SPL development and that our results are encouraging since no significant performance penalties were observed.

Keywords: Model-based Specification, Requirements Specification, Software Product Lines, Variability Modelling, Software Verification, Consistency Checking.

1. Introduction

A *Software Product Line* (SPL) is "a set of software–intensive systems sharing a common, managed set of features that satisfy the specific needs of a particular market segment or mission and that are developed from a common set of core assets in a prescribed way" [1]. SPLs are described in terms of *features* that express product functionalities concisely [2]. There are *common features* between products

*Principal corresponding author
Email addresses:
mauricio.alferez@campus.fct.unl.pt (Mauricio Alférez), roberto.lopez@jku.at (Roberto E. Lopez-Herrejon), amm@fct.unl.pt (Ana Moreira), vasco.amaral@fct.unl.pt (Vasco Amaral), alexander.egyed@jku.at (Alexander Egyed)

in a SPL (known as *commonalities*), and there are *variable features* that allow distinguishing between products in a SPL (known as *variabilities*). A common model to represent commonality and variability is the *feature model* [3] that represents the different features available in an SPL and describes their interdependencies.

To create a product from an SPL, feature realizations such as use cases, component diagrams, or code modules have to be composed according to a specific selection of features from a feature model. This composition process requires a mapping between the features and the artefacts that realize them. Although a number of different approaches have been proposed to create such mappings [4, 5, 6, 7], ensuring the consistency between features and their realizations has not been thoroughly researched for model-based SPL specifications.

Thus, in the context of this paper, *consistency checking* refers to the verification of consistency conditions of the relationships between the feature model and early model-based feature realizations, such as requirements and architecture specifications. Such relationships take into account dependencies and incompatibilities between model fragments, and dependencies and incompatibilities between features. In an inconsistent SPL, valid models may be invalid in terms of the feature model and invalid models may be valid in terms of the feature model. According to Apel et al., [8] the latter case is difficult to address because it is usually detected only after generating specific product variants based on a selection of features. However, due to the possible very large number of different valid features' selections and their relationships with model fragments, consistency checking for every product is feasible only for small SPLs [8]. Therefore, it is essential to check the entire SPL specification against the feature model to guarantee that any possible selection of features in the feature model renders a valid product variant.

The goal of this paper is to enable consistency checking between feature models and entire model-based SPL specifications, not only for a single product. The resulting approach employs propositional formulas that are produced based on the relationships between: (1) *domain constraints* that can be obtained from the SPL feature model, and (2) the relationships between fragments in model-based specifications and feature expressions (i.e., combinations of features and logical operators). Checking if all the products in an SPL satisfy consistency constraints is based on searching for a satisfying assignment of a propositional formula. Therefore, the VCC tool translates propositional formulas that can be evaluated by *satisfiability* (SAT) solvers [9]. For constraints that are not satisfied by the SPL, the VCC tool presents to the developer the features and fragments in the model-based specifications involved in the violation of the constraint. Such output is useful to take informed decisions about possible modifications and additions of domain constraints, model-based specifications, and their composition specification.

Our aim of checking consistency for an entire SPL instead of only for a single product is in line with other approaches [10, 11, 12, 13, 14]. The research area of these works is Generative Programming [3] applied to code-based modules. The goal of these approaches is to guarantee that every valid feature selection produces a type-correct program (see Section 6). Our work shares the same solution strategy of using propositional logic to encode the relationships and dependencies between features and artefacts. However, we focus on model-based specifications mainly used in early development of SPL such as requirements and architecture specification. Since our focus is on early development, the VCC approach does not assume a correct and complete feature model. Instead, it helps developers to complete the feature model based on the detected inconsistencies with respect to other models. The results of applying our approach to two case studies from different application domains are encouraging as no significant performance penalties were observed.

2. Background and Motivation

This section introduces the models we use during consistency checking. After this, it presents a motivating example that illustrates inconsistency based on these models.

2.1. Models Involved in Consistency Checking

Let us introduce first the models we use: (1) feature models, (2) model-based specifications, (3) mapping model between features and model-based specifications, and (4) composition specification model.

Feature Model. A feature model describes a set of all possible valid product configurations [3]. A feature model configuration specifies a concrete product in terms of its features. There are different notations to specify feature models and most of them can be translated to propositional formulas. A formula contains, for each feature, a boolean variable and expresses the constraints between features. In each formula the standard $\wedge$ (and), $\vee$ (or), $\neg$ (not), $\rightarrow$ (implies), and $\leftrightarrow$ (double implication) operators of propositional logic can be used.

Model-based Specifications. Features can be realized by other models such as requirements analysis models, architectural models or code-based modules. In this section we employ use scenarios as an example of model-based specification for feature models. Use scenarios are a widely used technique that describes, step by step, how an actor intends to use a system [15]. We employ use cases and activity diagrams to describe use scenarios as they are commonly used in mainstream UML-based methods such as RUP [16]. Thus, in contrast to mere free-form ambiguous textual scenario descriptions, use case and activity diagrams conforming to the UML language help to reduce ambiguity in the specifications [2]. Section 3 presents how VCC uses this and other kinds of realizations such as architectural components diagrams.

Composition Specification. A way to produce specific product models is through a composition process, also called, derivation process. Without composing the models for all the possible product variants VCC guarantees that a feature model and the models that specify its features are consistent. Thus, in this paper we show composition of model-based specifications only to illustrate inconsistencies when producing product variants in our motivating example. We use the Variability Modelling Language for Requirements (VML4RE) [5, 17] as an example of a composition specification language and derivation tool that is appropriate to customize use scenarios. We employ VML4RE because it offers an easy reference to model elements and their transformation. Those model transformations usually are not as complex as typically sophisticated model transformations created by general purpose model transformation languages such as ATL [18], AGG [19] and QVT [20].

Mapping Model. Derivation of products in an SPL requires a mapping between features from a feature model and their feature realizations (e.g., use cases and component diagrams, or code modules). A number of different approaches have been proposed to create mappings among features and feature realizations [4, 5, 6, 7]. Additionally, feature model editors and supporting tools such as pure::variants[1], Feature Mapper[2], and FeatureIDE[3] support the mapping activities. The mapping is usually accomplished using annotation tables containing pairs of "feature" and "related asset" or presence conditions attached to parts of the assets such as code annotations, model stereotypes and notes.

2.2. Motivating Example

Creating a Feature Model. Figure 1 (a) shows part of a feature model for one of our case studies, the *Smart Home* SPL [21]. Smart Home has four optional features, AUTOMATED WINDOWS, AUTOMATED HEATING, REMOTE HEATING CTRL and INTERNET to control the heater and other devices remotely. Also, it has a set of common features, such as MANUAL WINDOWS, MANUAL HEATING and INHOME SCREEN that will be included in all the target products produced from the Smart Home SPL. Specific product configurations can be defined by selecting optional features in the feature model. Figure 1 (b) shows a sample product configuration of the Smart Home SPL called PRODUCT-1 that includes all the features. Figure 1 (c) shows another sample product configuration called PRODUCT-2 that has all features except AUTOMATED WINDOWS, and that will be used to illustrate consistency problems between features and use scenarios. Domain constraints in the feature model such as the REQUIRES relationship from REMOTE HEATING CTRL to INTERNET, can be added incrementally when realizations are created (discussed below).

Creating Model-based Specifications. Use cases and activity diagrams provide a description of what products in the domain should do. Feature models determine which functionality can be selected when engineering new products from the SPL. Therefore, product requirements specifications consist of customized use cases and activity diagrams. The customization is guided by a composition specification discussed in the next subsection.

[1] http://www.pure-systems.com/
[2] http://featuremapper.org/
[3] http://wwwiti.cs.uni-magdeburg.de/iti_db/research/featureide/

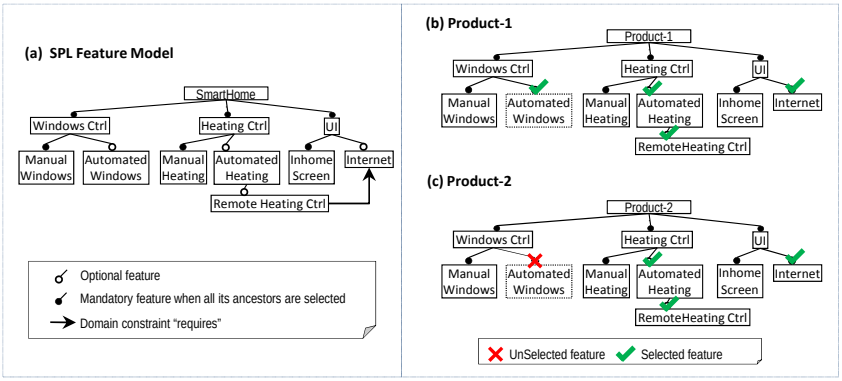


Figure 1: (a) Simplified sample of the Smart Home feature model; (b) Sample configuration that includes all features; and (c) Sample configuration that excludes the Automated Windows feature.

Figure 2 (a) (left and right-hand sides) shows two of the models that are part of the realizations for PRODUCT-1. Figure 2 (a) (left-hand side) shows an activity diagram that depicts the possible scenarios for the use case CTRLTEMPREMOTELY that is depicted in the use case diagram of Figure 2 (a) (right-hand side). The activity diagram also comprises activities of use cases OPENANDCLOSEWINAUTO, ADJUSTHEATERVALUE and NOTIFYBYINTERNET. Within this activity diagram it is possible to select several scenarios that correspond to different paths. Two of all the possible scenarios are: Scenario (S1) includes reaching the in-home temperature and save energy by means of closing some windows, and Scenario (S2) to use the heater to reach the desired in-home temperature.

Figure 2 (a) (right-hand side) shows part of the use case model composed for PRODUCT-1. The INCLUDES relationship describes the case where one use case, the *base* use case, includes the functionality of another use case, the *inclusion* use case. The INCLUDES relationship supports the reuse of functionality in a use case diagram and is used to express that the behavior of the *inclusion* use case is common to other use cases. Note that INCLUDES relationships between use cases may constrain the relationship between the features related to them. For example, the INCLUDES relationship between the base use case CTRLTEMPREMOTELY that includes the use case OPENANDCLOSEWINAUTO may imply that feature REMOTEHEATINGCNTRL requires the feature AUTOMATEDWINDOWS.

It is important to note that the customization of realizations such as activity diagrams and scenarios depends on the features chosen for the SPL product and also on the relationship with the use case diagram. For example, given that in PRODUCT-2 the feature AUTOMATEDWINDOWS was not selected, the WINACTUATOR actor in the use case diagram as well as the swimlane (also called activity partition) related to WINDOWSACTUATOR will not appear in any diagram. Therefore, scenarios such as Scenario (S1) described in a previous paragraph, are not realizable due to the lack of windows actuators. This and other constraints will be discussed in Section 3.

```
1.  Variant { name : "A-W" for : "Automated Windows"
2.    + UseCase : "OpenAndCloseWinAuto" in Package : "WinMan"
3.    + Partition : "WinActuator" in ActDiagram :"CtrlTempAuto"
4.    ...
5.  }
6.  Variant { name : "R-H"
7.    for : And ("Remote Heating Ctrl","Automated Heating","Internet")
8.      + UseCase : "CtrlTempRemotely" in Package : "Heating"
9.      + UseCase : "CalcEnergyConsumption" in Package : "Heating"
10.     Includes from UseCase : "CtrlTempRemotely"   to UseCase(s) :
11.           "NotifyByInternet" and "OpenAndCloseWinAuto" and "AdjustHeaterValue"
12.     ...
13. }
    ...more variants
```

Figure 3: Composition specification of variants A-W and R-H.

Creating a Composition Specification. Figure 3 illustrates a composition specification that guides the specification of the transformation of requirements specifications of products in the SmartHome SPL. VML4RE [5, 17] is a textual language that allows associating *actions*, that wrap a set of model trans-

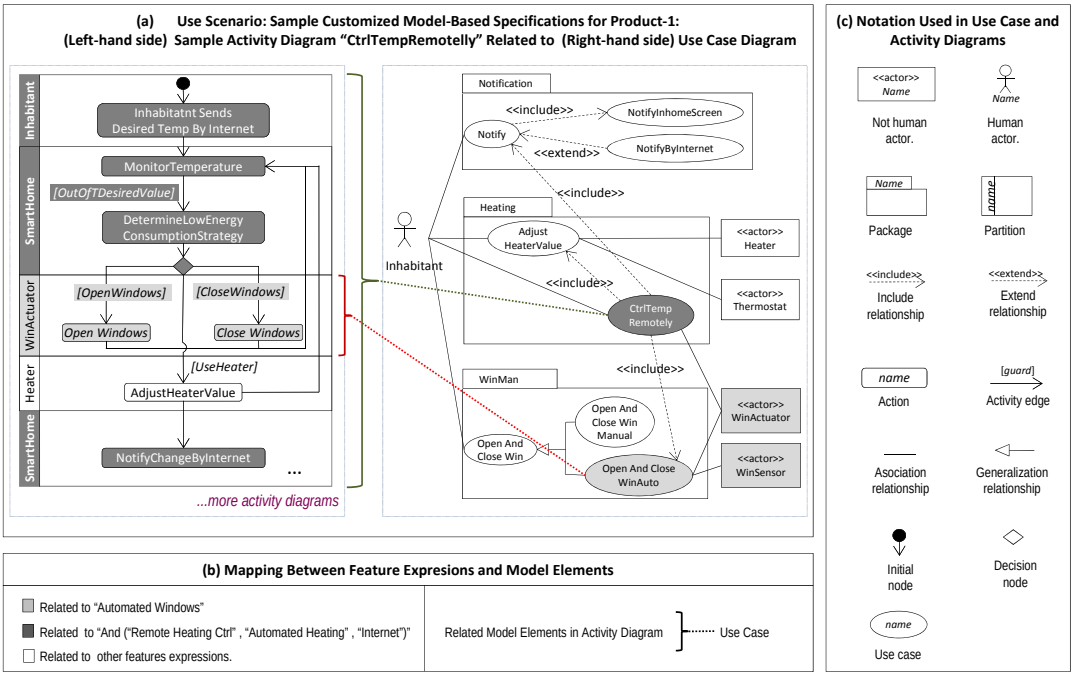


Figure 2: (a) Sample customized realizations for PRODUCT-1; (b) Mapping between feature expressions and model fragments; and (c) Notation used in use case and activity diagram in (a).

formations for specific requirements models such as use cases and activity diagrams, to combinations of features written as logic expressions that we call *feature expressions*. Feature expressions can be:

- *Atomic*, representing single features such as "Automated Windows" in Figure 3, Line 1, and
- *Compound*, containing logic operators such as AND, NOT and OR such as "And ("Remote Heating Ctrl","Automated Heating","Internet")" in line 7 of Figure 3.

Feature expressions evaluation works as follows: if in a feature model configuration a feature is selected to be part of a product, that feature evaluates to TRUE and if the feature is not selected it evaluates to FALSE. Thus, a feature expression can be evaluated to TRUE or FALSE taking into account the Boolean value of each feature in the feature expression. If a feature expression evaluates to TRUE its corresponding actions will be processed and applied to the base model. Otherwise, if the feature expression evaluates to FALSE, the next feature expressions will be read and evaluated until the end of the composition specification.

In our example if AUTOMATEDHEATING,REMOTE HEATING CTRL, AUTOMATED HEATING and INTERNET features are selected in a product configuration, the feature expression associated to the variant named "R-H" (i.e., the compound feature expression: And ("Remote Heating Ctrl","Automated Heating","Internet")) will be evaluated to TRUE. The consequence is that the actions inside the "R-H" variant block (Figure 3, lines 6-13) will be processed and applied to a base model. For example, the CTRLTEMPREMOTELY use case will be inserted into the package HEATING and then it will be related to other use cases using INCLUDES and EXTENDS relationships. If more than one feature expression is evaluated to TRUE, the default composition order follows a top-down sequence. Note that for simplicity of explanation we omitted in Figure 3 some of the actions, such as , the insertion of some actors such as WINSENSOR and WINACTUATOR and some partitions such as HEATER.

Creating a Mapping Specification. Figure 2 (a) shows use case and activity diagrams fragments, such as actors and use cases, related with the variants shown in Figure 3. The base mechanism to relate parts of the realizations to features is to use a correspondence table (or mapping table), as presented by [4, 5, 8, 7]. It is possible to parse the

composition specification to generate the mapping between variants and parts of the realizations (more details in Section 3.3). Therefore, for example, if the variant A-W inserts the OpenAndCloseWinAuto use case, we relate the feature expression of A-W ("automatedWindows") to OpenAndCloseWinAuto. To facilitate the visualization of such relationships with the models, we assign different gray tones to the model elements according to the feature expressions of variants that they are related to (see mapping in Figure 2 (b)). Also, note that specific model fragments could be related to more than one variant; this may be considered as a M-to-N (where M, N >= 1) mapping between feature expressions of variants and model fragments (not illustrated in Figure 2).

2.3. Consistency Checking Motivation

Consistency checking aims at ensuring that requirements and constraints related to the feature model and model-based specifications are consistent between them. Let's analyse two requirements in the Smart Home example. Requirement-1, which is inferred from the feature model and the feature expressions in the mapping model, and Requirement-2 which is inferred from the model-based specifications:

***Requirement-1*:** *Only one, none or both* R-H *and* A-W *variants can be included in a product.* This information is inferred from the feature model and mapping model because all the features in the feature expression of the R-H variant are optional and not exclusive between them (i.e., Remote Heating Ctrl, Automated Heating and Internet are optional features), and the only feature in the feature expression of variant A-W is also optional (i.e., the Automated Windows feature is optional).

***Requirement-2*:** *If the use case* CtrlTempRemotely *is provided in a product, then the use case* OpenAndCloseWinAuto *and its related steps in the activity diagram must be provided too in order to support all the intended use scenarios.* This is implicit in the includes relationship from the use case CtrlTempRemotely to OpenAndCloseWinAuto in the use case diagram in Figure 2 (a) (right-hand side) and also because of the control flow between: "Determine Low Energy Consumption Strategy" and "Open Windows" / "Close Windows " in Figure 2 (a) (left-hand side).

In the particular case of the Smart Home use scenarios, an inconsistency can be detected: there is at least one product that cannot satisfy Requirement-1 and Requirement-2. Let's analyse how both requirements are satisfied or not in each product.

While Requirement-1 is satisfied by all the product configurations, therefore satisfied by Product-1 and Product-2, since they have the same feature model and mapping models, Requirement-2 is satisfied by Product-1 only as its use cases and activities supported all the required use scenarios for this product. For example, given that the *base* use case CtrlTempRemotely was provided in Product-1, the use cases related to it through an Includes relationship, for example OpenAndCloseWinAuto, are also present in the model. The *include* relationship supports the reuse of functionality in use case diagrams in which one use case (the *base* use case) requires the functionality of another use case (the *inclusion* use case). Therefore, all possible use scenarios related to CtrlTempRemotely are supported only when its *inclusion* use cases are included.

Requirement-2 is not satisfied by Product-2 because its feature configuration (shown in Figure 1 (c)) does not include the Automated windows feature. Therefore, the feature expression of variant A-W (i.e., Automated windows)(Figure 3, Line 1) evaluates to FALSE and the actions inside its variant block are not processed, for example, the inclusion of the use case OpenAndCloseWinAuto. The result is that the functionality provided by OpenAndCloseWinAuto will not be present in the requirements of Product-2 and therefore it will not be taken into account in later stages of its development process, thus given no support for the scenarios related to CtrlTempRemotely.

One solution to solve the inconsistency for our example is to guarantee the presence of the feature Automated Windows when Automatic Heating or Remote Heating Ctrl are selected, in every possible feature model configuration. This can be achieved in any of two ways (1) creating one domain constraint requires from Remote Heating Ctrl to Automated Windows or (2) to establish that Automated Windows is a mandatory feature. However, the number of possible feature combinations may grow exponentially with the number of features of the SPL and makes unfeasible

to manually check the consistency of all products, one by one.

To guarantee that all the products derived from a feature model have consistent specifications, we take into account the relationships between constraints specified in feature models and its model-based specifications. We will discuss this in the next section.

3. Ensuring Consistency between Feature Models and Model-Based Specifications - The VCC Approach

3.1. Overview

Figure 4 presents an overview of the VCC approach. Section 2 explained and exemplified the activity "Create or Modify Models": *Feature Model*, *Model-based Specifications* (during this section we called them *Realizations* for short) as well as its corresponding *Composition Specification*.

Domain Constraints are denoted as DC_f and are translated from a *Feature Model*[4]. Section 3.2 explains how to obtain and express DC_f in terms of features (f).

Realization Constraints are denoted as RC_f and are obtained from:

- *Dependencies and Incompatibilities* between model elements in realizations (Section 3.4), and
- *Mapping* between variants and model elements in realizations (Section 3.3).

Section 3.5 explains how to obtain RC_f from the *Mapping*, and *Dependencies and Incompatibilities* in terms of features.

We employ propositional logic to express and relate DC_f and RC_f. We want to guarantee that *Domain Constraints* meet *Realization Constraints* (i.e., $RC_f \rightarrow DC_f$) and that *Realization Constraints* meet *Domain Constraints* (i.e., $DC_f \rightarrow RC_f$).

A *Consistency Checking Report* is the output of the process. To *Check Consistency,* Equation 1 should not be "satisfiable" for each and every *consistency rule* that we want to check. If it is satisfied, it means that domain constraints do not meet the realizations constraints (i.e., $\neg\,(RC_f \rightarrow DC_f)$ is satisfiable), or that the realizations constraints do not meet domain constraints (i.e., $\neg\,(DC_f \rightarrow RC_f)$ is satisfiable).

$$\neg\,(RC_f \rightarrow DC_f) \bigvee \neg\,(DC_f \rightarrow RC_f) \quad (1)$$

Section 3.6 explains how VCC and a SAT solver can identify the inconsistent domain constraints and the fragments of the associated models that realize them. This information can support the developer to take informed decisions on modifications of the initial SPL models, for example, creating or modifying domain constraints or relationships between model elements.

It is important to note that feature model editors and supporting tools such as SPLOT[5], pure::variants , Feature Mapper , and FeatureIDE support two activities of VCC "Obtain Mapping between Variants and Realizations" and "Obtain Constraints between Features". For example, SPLOT generates a propositional formula (i.e., the domain constraints DC_f) from feature models and Feature Mapper helps to relate model fragments to feature expressions (i.e., the *Mapping*).

Section 2.1 shows that at least one product (i.e., PRODUCT-2) from the Smart Home SPL is inconsistent. In that case, constraints that can be inferred from the realizations between elements in use scenarios, such as the INCLUDES between use cases and related control flows, and actions in activity diagrams, imply the application of domain constraints (i.e., $(RC_f \rightarrow DC_f)$). The particular consistency condition that guarantees that $\neg\,(RC_f \rightarrow DC_f)$ be not satisfiable is defined by the rule (more rules in Section 5):

- *Required Inclusion Use Case:* at least one variant defining an inclusion use case must be selected in every feature configuration containing the variant that introduces a base use case linked to the inclusion use case.

3.2. Obtain Constraints between Features (DC_f)

The constraints between features (domain constraints) are obtained from an SPL feature model as shown in Figure 4. Therefore DC_f is the same for all possible product configurations. DC_f is obtained from a translation table between feature

[4]There are several works in the area of feature model verification and analysis that can be used to derive domain constraints from feature models (for a comprehensive survey see [22]).

[5]http://www.splot-research.org/

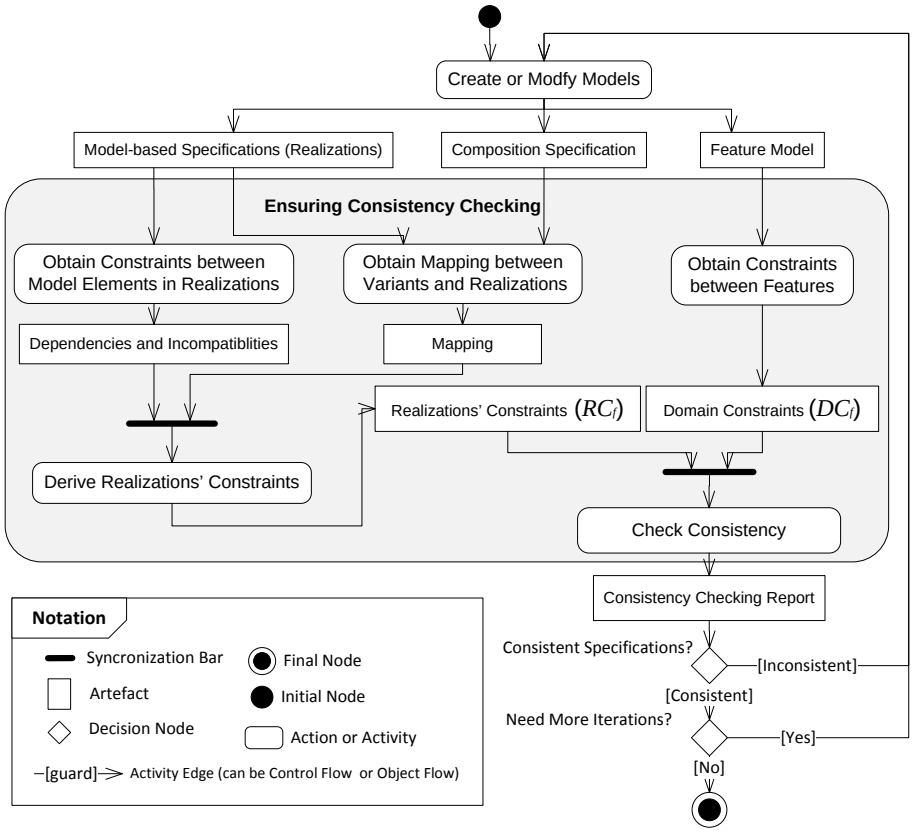


Figure 4: Overview of the Variability Consistency Checking (VCC) approach.

model elements and propositional formulas with its corresponding conjunctive normal form (CNF[6]) which is the standard input form for satisfiability solvers.

The idea of translating feature models to propositional logic was proposed by Mannion et al. [23] and Batory et al. [24] for example. Based on these works Table 1 shows the translation between feature model elements and propositional logic obtained from the following steps [22]:

1. Each feature maps to a variable of the propositional formula,
2. Each relationship, e.g., requires, optional, is mapped into one or more formulas,
3. The resulting formula is the conjunction of all the resulting formulas of step 2 plus an additional constraint assigning TRUE to the variable that represents the root feature (e.g., $SmartHome \longleftrightarrow TRUE$).

The right-most column of Table 1 is the translation to CNF form that will be used by the SAT solver[7]. The mapping rules from propositional logic to CNF form are based on classical logical equivalences [25] such as the double negative law, De Morgan's laws, and the distributive law.

Equation 2 shows an example for the HEATING CTRL branch, the most complex branch in Figure 1 (a), and relates directly with the example "Required Inclusion Use Case" rule. The translation obtained in the first line of Equation 2 means that all products unconditionally must contain the root feature SMART HOME. Line 2 means HEATING CTRL must be included in all the products, since it is a mandatory feature. Line 3 means that MANUAL HEATING is included in all the products that include their

[6] A formula is in CNF form when it is a conjunction of clauses and each clause is a disjunction of literals. CNF formulas have special characteristics: i) a literal and its complement cannot appear in the same clause, ii) the only connectives are: AND ($\wedge$), OR ($\vee$), and NOT($\neg$), and iii) NOT($\neg$) can only be used as part of a literal.

[7] The characters representing the logical operators may change according to the specific SAT solver tool used.

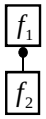
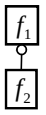
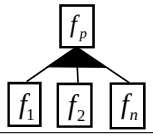
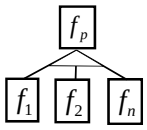
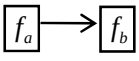
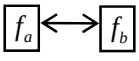
Relationship		Propositional Logic
Mandatory		$f_1 \leftrightarrow f_2$
Optional		$f_2 \rightarrow f_1$
Or		$f_p \leftrightarrow (f_1 \vee f_2 \vee ... \vee f_n)$
XOr		$(f_1 \leftrightarrow (\neg f_2 \wedge ... \wedge \neg f_n \wedge f_p)) \wedge$ $(f_2 \leftrightarrow (\neg f_1 \wedge ... \wedge \neg f_n \wedge f_p)) \wedge$ $(f_n \leftrightarrow (\neg f_1 \wedge \neg f_2 \wedge ... \wedge \neg f_{n-1} \wedge f_p))$
Requires		$f_a \rightarrow f_b$
Excludes		$\neg(f_a \wedge f_b)$

Table 1: Mapping feature models to propositional logic and CNF (based on [25, 22, 24]).

parent feature (i.e., HEATING CTRL), in contrast to AUTOMATED HEATING and REMOTE HEATING CTRL (lines 3-4), that may be, or not, included when their respective parents HEATING CTRL and AUTOMATED HEATING are included in a product. Line 5 means that REMOTE HEATING CTRL *requires* the INTERNET feature.

$$
\begin{aligned}
&1.\ (SmartHome \leftrightarrow TRUE) \wedge \\
&2.\ (SmartHome \leftrightarrow HeatingCtrl) \wedge \\
&3.\ (HeatingCtrl \leftrightarrow ManualHeating) \wedge \\
&\quad\ (AutomatedHeating \rightarrow HeatingCtrl) \wedge \\
&4.\ (RemoteHeatingCtrl \rightarrow AutomatedHeating) \wedge \\
&5.\ (RemoteHeatingCtrl \rightarrow Internet)
\end{aligned}
\tag{2}
$$

This Section addresses DC_f of Equation 1. The following sections show how to obtain RC_f, which comprises a set of constraints that are essential for consistency between realizations and domain constraints such as the ones expressed in Equation 2.

3.3. Obtain Mapping between Variants and Realizations

To obtain *Realizations' Constraints* (RC_f) it is necessary to know two kinds of relationships as shown in Figure 4: (1) *Mapping* relationships between each variant (each one with an assigned feature expression) and the model elements that depend on the variant, and (2) *Dependencies and Incompatiblities* between model elements of the realizations. Figure 5 shows in red and dashed outline the VCC metaclasses that help to support the activity "Obtain Mapping between Variants and Realizations". Also, Figure 5 shows in blue and dotted outline the metaclasses that help to support the activity "Obtain Constraints between Model Elements in Realizations", explained in the next section.

The VCC metamodel of Figure 5 is independent of the concrete syntax employed to represent concrete instances of MODELELEMENT metaclass. Our work focuses on early development of SPL and employs Use Cases, Activity and Component diagrams therefore, Figure 6 shows only some of the metaclasses that extend the MODELELEMENT metaclass of UML that are related to these diagrams (e.g., COMPONENT, INTERFACE, ASSOCIATION, ACTION, USECASE, ACTOR, ACTIVITY, PARTITION and PACKAGE). As part of our future work we will research how well VCC can be applied to not UML-based models.

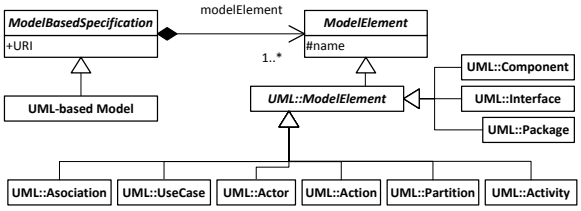


Figure 6: Example of the specialization of the abstract metaclasses MODELELEMENT and MODELBASEDSPECIFICATION with an UML-based model and concrete MODELELEMENT metaclasses.

To derive realization constraints it is necessary to map variants to concrete instances of model elements in the model-based specifications. In Figure 5 a VARIANT in VCC has a name and one feature expression. In VCC we generalize the relationship between variants and model elements using a metaclass MAPVARME. The mapping can be

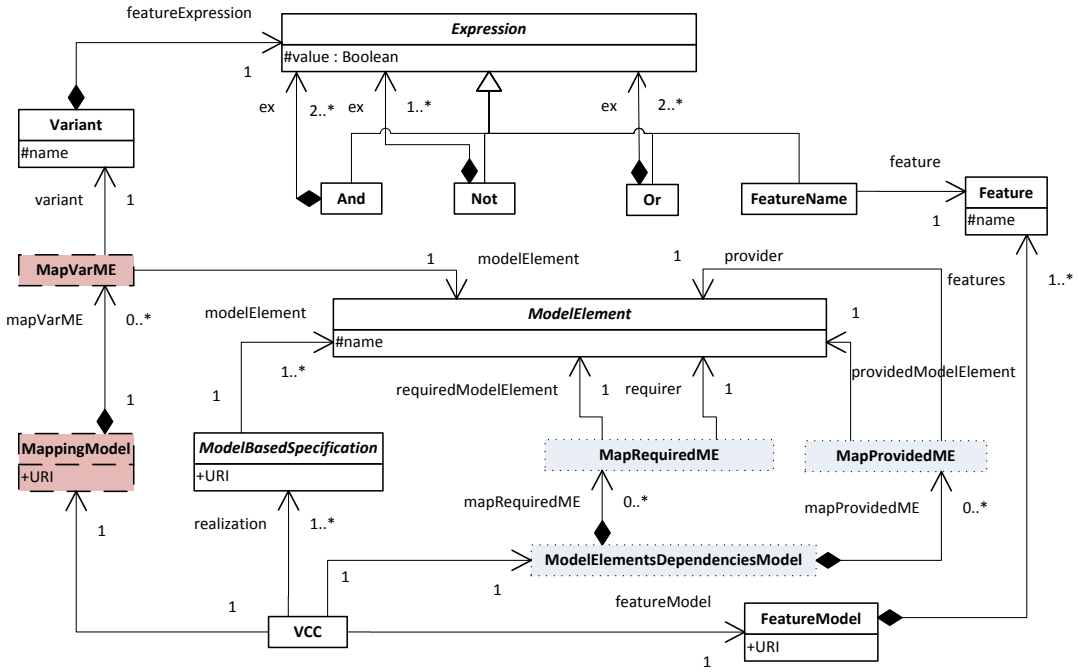


Figure 5: VCC metaclasses related to mappings between model elements (blue colour and dotted outline) and between model elements and variants (red blue colour and dashed outline).

done instantiating MapVarMe and assigning existing model elements and variants to it.

There are two ways to instantiate the mapping model between variants and model elements:

1. *Manually.* Each instance of MapVarMe relates an existing variant with one model element. Modelling tools such as EMF[8] provide libraries with specialized methods. For example, given a metamodel EMF provides interfaces and a factory to create objects that represent instances of each metaclass. The requirements to create a mapping model are the elements that will be mapped. Thus, we need (1) to create a model-based specification (e.g., use case, activity and component diagrams), and (2) one or more instances of variants (i.e., objects of the metaclass Variant). The steps are the following:
   - Create an instance of the MappingModel metaclass.
   - Create one or more instances of MapVarMe. Each instance will contain a pair of references (Variant, ModelElement).
   - Add all the instances of MapVarMe created in the previous step to the empty list of mappings of the MappingModel.

   To facilitate the creation of the mapping, developers can employ model creation wizards and model editors offered by modelling frameworks such as EMF.
2. *Automatically, based on the actions of the composition specification.* Part of the mappings can be discovered reading all the variants with their contained actions. Figure 7 shows that each variant contains any number of actions in a composition specification. Abstract Action types, such as Insert and Connect (shown in yellow and dashed outline), have to be specialized by some metaclasses (shown in green and dotted outline), for example, InsertUseCase, ConnectByIncludes and InsertComponent (shown in green and dotted outline). Then, for actions of type Insert, VCC creates an instance of MapVarMe and assigns to it the variant and the inserted model elements.

[8] http://www.eclipse.org/modeling/emf/

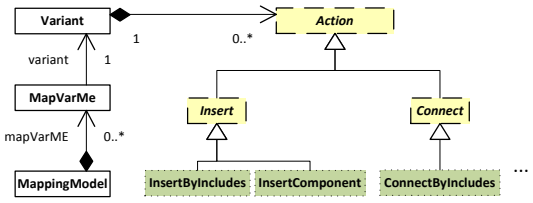


Figure 7: Abstract metaclasses (shown in yellow and dashed outline) and some of their specializations (shown in green and dotted outline) that help to infer some mappings between variant and model elements.

Figure 8 shows an example based on Line 2 of the composition specification of Figure 3 also shown in Figure 8 (c). We will obtain an instance of MapVarMe (Figure 8 (b)) that references the variant with name A-W and the model element OpenAndCloseWinAuto of type UseCase.

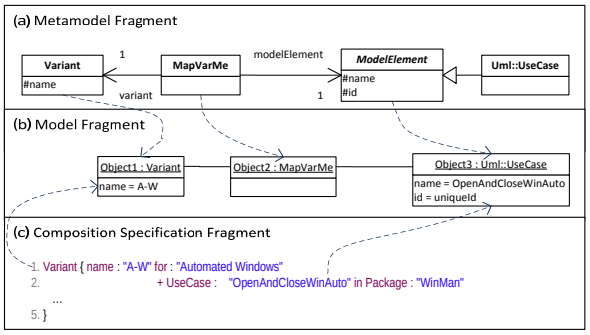


Figure 8: (a) VCC metaclasses related to mappings between model elements and variants; (b) Object diagram of the mapping generated for the composition specification fragment shown in (c); and (c) Composition specification fragment from the example.

3.4. Obtain Constraints between Model Elements in Realizations

The process in Figure 4 shows that one of the inputs for the activity "Derive Realizations' Constraints" is the list of dependencies between model elements in feature realizations. The requirement to create an instance of ModelElementsDependenciesModel is to have a model-based specification (e.g., use case, activity and component diagrams). There are three not mutually exclusive ways of obtaining dependencies between model elements:

1. *Manually, instantiating the metaclasses* MapRequiredMe *and* MapProvidedMe. Each instance of MapRequiredMe relates one required model element with one of the model elements that requires it. Also, it is possible to instantiate the metaclass MapProvidedMe to relate one model element with one of the models that it provides. The concept of "provides" and "requires" is similar to the one used in component-based development, where each component provides (i.e., realizes) some interfaces or requires (i.e., uses) some other interfaces. In VCC we generalize this concept so it can be applied for any kind of model element. The steps required to create a model elements dependencies model are:
   - Create an instance of the ModelElementsDependenciesModel metaclass.
   - Create one or more instances of MapProvidedMe and add them to the ModelElementsDependenciesModel. Each instance will contain a pair of references (Provider, ProvidedModelElement).
   - Create one or more instances of MapRequiredMe and add them to the ModelElementsDependenciesModel. Each instance will contain a pair of references (Requirer, RequiredModelElement).
2. *Automatically instantiating* MapRequiredMe *and* MapProvidedMe *based on the actions of the composition specification.* All the actions that suggest the pre-existence of model elements are good candidates to be analysed. This is the case presented in Figure 9 (b) where the use case CtrlTempRemotely includes the behavior represented by OpenAndCloseWinAuto. Therefore, the requirer is CtrlTempRemotely and the required model element is OpenAndCloseWinAuto.
   Obtaining dependencies based on actions that are read from the composition specification is supported by the specializations of the metaclass Action shown in Figure 5. The following steps should be followed to create a model elements dependencies model:
   - Create an instance of the ModelElementsDependenciesModel metaclass.

- For each instance of ConnectAction create one instance of MapRequiredMe and add them to the ModelElementsDependenciesModel. Each instance will contain a pair of references (Requirer, RequiredModelElement).

Figure 9 shows an example of the relationships between model elements and their metaclases based on a line of a composition specification written using VML4RE (Figure 9 (c)). ConnectByIncludes in Figure 9 (a) is a metaclass in VML4RE that represents the action (Figure 9 (c)) that links from a use case to one or more use cases using the Include relationship as it is shown in Figure 9 (b). ConnectByIncludes is one of the actions that specializes the action Connect from the VCC metamodel shown in Figure 5, and UseCase is a type of ModelElement referenced by ConnectByIncludes that is defined in UML.

3. *Automatically instantiating* MapRequiredMe *and* MapProvidedMe *based on the realization models.* This is probably the best way to derive dependencies when there is explicit information about the elements provided and required. This is the case of UML component diagrams where each component declares explicitly a list of provided and required interfaces. Programmatically, it is straightforward to parse a realization model and create instances of MapProvidedMe and MapRequiredMe. Thus, each instance will relate a component that plays the role of either provider or requirer, and an interface that plays the role of provided or required model element, respectively. In this case, the steps required to create a model elements dependencies model are the following:
   - Create an instance of the ModelElementsDependenciesModel metaclass.
   - For each instance of ModelElement that plays the role of provider create an instance of MapProvidedMe and add it to the ModelElementsDependenciesModel. Each instance will contain a pair of references (provider, providedModelElement).
   - For each instance of ModelElement that plays the role of provider create an instance of MapRequirerMe and add it to the ModelElementsDependenciesModel. Each instance will contain a pair of references (requirer, requiredModelElement).

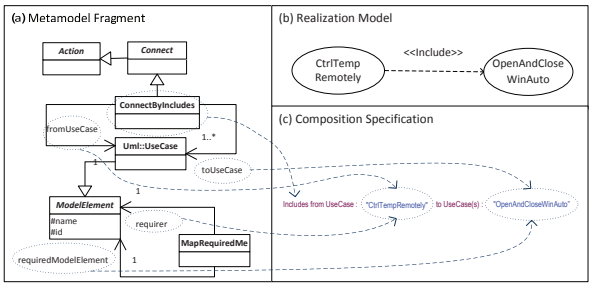


Figure 9: (a) Set of metaclases that support the dependencies for the ConnectByIncludes action; (b) Exemplar model fragment related to an Includes relationship between use cases; and (c) the composition specification

3.5. Deriving Realizations' Constraints (RC_f)

Realizations' constraints act as consistency rules describing the relationships that must hold among the different model elements. There are two basic types of realization constraints that we classify according to the type of domain constraint that they relate with: (1) constraints that imply an excludes relationship between features, therefore implying incompatibilities between variants, and (2) constraints that imply a Requires relationship between features that therefore imply dependencies between variants. This section shows the equations that express each of these. In the definition of the equations, the function F receives as input a variant and returns its corresponding feature expression. Thus, we can express the realizations' constraints in terms of feature expressions and not in terms of their related variants.

Excludes Relationship*:* Let Var be a variant that defines a model element e. Another variant $VarExc_i$ conflicts with Var if $VarExc_i$ defines a model element c which cannot be present in the same realization of a product where element e is. Due to the incompatibility between the elements e and c, if variant Var is selected then variant $VarExc_i$ should not be selected for the same product configuration. This is denoted in Equation 3, where n represents the total number of conflicting variants:

$$RC_{f,\,Exc} \equiv F\,(Var) \rightarrow \neg \bigvee_{i=1}^{n} F\,(VarExc_i) \tag{3}$$

Requires Relationship*:* Let Var be a variant that refers to a model element e defined by another variant. To be consistent, the model-based specifications of a product that includes Var must also include at least one other variant $VarReq_i$ (required variant) where element e is defined. This is denoted in Equation 4, where n represents the number of required variants:

$$RC_{f,\,Req} \equiv F\,(Var) \rightarrow \bigvee_{i=1}^{n} F\,(VarReq_i) \tag{4}$$

Equation 4 evaluates to FALSE when any action in variant Var requires an element or set of *required elements* that are not composed, for example. This happens because none of the corresponding variants $VarReq_i$ that introduce the *required elements* was selected in the product configuration. Also, expression 3 evaluates to FALSE when variant Var defines an element or set of elements that are introduced in the realizations that also contain conflicting elements defined by other variant(s) $VarConf_i$.

The rule "Required Inclusion Use Case" mentioned at the beginning of this section is an example of $RC_{f,\,Req}$. An instance of this constraint is found in our motivational example related to the use scenario of CntrlTempRemotely. For example, given that variant Var = R-H is selected (i.e., a product with Remote Heating Cntrl, Automated Heating and Internet features) and it is related to the base use case CtrlTempRemotely, we want to guarantee that there is at least one variant (e.g., $VarReq_i$ = A-W) related to the inclusion use case OpenAndCloseWinAuto (i.e., model element e = use case OpenAndCloseWinAuto) and that its feature expression evaluates to TRUE in all possible feature model configurations. In this way we guarantee the presence of the functionality required by CtrlTempRemotely, such as to include a WindowsActuator that regulates the temperature opening and closing windows. Hence, with that information, it is possible to create the realization constraint $RC_{Var} \equiv$ R-H $\rightarrow$A-W. We can get a constraint instance replacing the variants by their corresponding feature expressions using function F to obtain Equation 5:

$$RC_{f,\,Req} \equiv F(R-H) \rightarrow F(A-W) \equiv (\,RemoteHeatingCntrl \wedge AutomatedHeating \wedge Internet\,) \rightarrow (\,AutomatedWindows\,) \tag{5}$$

3.6. Check Consistency

The "Check Consistency" activity depicted in Figure 4 depends on the kind of constraint created in the previous section. If we replace RC_f of Equation 4 in Equation 1 and perform some logic manipulation we obtain the expression in Equation 6 (for a step by step process see the Appendix section).

$$\begin{array}{c} Realization\,Constraint\,of\,type\,Requires: \\ \neg\,(RC_{f,\,Req} \rightarrow DC_f) \bigvee \neg\,(DC_f \rightarrow RC_{f,\,Req}) \\ \equiv \left(\left(\neg F\,(Var) \bigvee_{i=1}^{n} F\,(VarReq_i)\right) \wedge \neg DC_f\right) \\ \bigvee \left(DC_f \wedge F\,(Var) \wedge \bigwedge_{i=1}^{n} \neg F\,(VarReq_i)\right) \end{array} \tag{6}$$

Similarly, if we replace RC_f of Equation 3 in Equation 1, and perform some logic manipulation, we obtain the expression in Equation 7.

$$\begin{array}{c} Ralization\,Constraint\,of\,type\,Excludes: \\ \neg\,(RC_{f,\,Exc} \rightarrow DC_f) \bigvee \neg\,(DC_f \rightarrow RC_{f,\,Exc}) \\ \equiv \left(\left(\neg F\,(Var) \vee \bigwedge_{i=1}^{n} \neg F\,(VarExc_i)\right) \wedge \neg DC_f\right) \\ \bigvee \left(DC_f \wedge F\,(Var) \wedge \bigvee_{i=1}^{n} F\,(VarExc_i)\right) \end{array} \tag{7}$$

The input for satisfiability checking are expressions such as the ones in Equations 6 and 7. Each expression to be checked is instantiated with:

- The specific domain constraints, DC_f of the SPL produced in Equation 2,
- The feature expressions related to the variants Var and either the set of its required variants $VarReq_i$ or the set of its conflicting variants $VarExc_i$.

To know which part of the disjuntion in Equation 6 and Equation 7 evaluates to TRUE (i.e., it is inconsistent), each term of the disjunction is evaluated separately. Therefore, we consider that the feature model and its realizations are consistent only when both terms of the disjunction evaluate to FALSE.

The possible results generated by a satisfability (SAT) checker for each expression (Figure 4 - "Consistency Checking Report")) can be TRUE (satisfiable) or FALSE (not satisfiable). If FALSE is obtained for all the expressions, we know that the SPL is consistent because there are no inconsistencies between the relationships and dependencies (e.g., excludes, optional, mandatory, requires) between features depicted in the SPL feature model, and their realizations. If TRUE is obtained, our tool, based on the mapping between variants and model elements in the realizations, shows a list of the variants and the model elements related to each inconsistency.

Taking the example of the Smart Home feature model depicted in Figure 1, the result of the SAT solver for rule "Required Inclusion Use Case" is that it is satisfiable (i.e., it evaluates to TRUE). This means that there is an inconsistency between the features and use scenarios. An example of the type of message generated by our tool to the user is (see tool support in Section 4):

> "...Inconsistent use scenario(s) [CtrlTempRemotelly] and feature(s) in feature expression(s) of variant(s) [A-W], [R-H]. The Action: [Includes from UseCase: "CtrlTempRemotely" to Use Case(s) "OpenAndCloseWinAuto"] implies a [Requires] relationship between variant [R-H] and required variant(s) [A-W] that is not enforced in the SPL feature model...".

Based on this information, for the SAT solver to evaluate to FALSE, developers may consider, for example:

- Modify the feature model: the set of SPL domain constraints that can be extracted from the feature model can be modified, for example by creating a REQUIRES relationship for AUTOMATEDHEATING feature to AUTOMATEDWINDOWS, or by changing the AUTOMATEDWINDOWS feature from optional to mandatory.
- Modify use scenarios and composition model: for our particular rule, developers may want to check if indeed the INCLUDES association between use cases CTRLTEMPREMOTELY and OPENANDCLOSEWINAUTO is mandatory for every single product, or not.

4. Tool Support

Tools for consistency checking can be highly effective for detecting errors in variability modelling. Such tools can find errors people miss, but they can also alleviate developers from the tedious and error-prone task of checking feature models and their realizations for consistency [26].

VCC [9] tool supports the process described in Section 3.1. Figure 10 shows the architecture of the VCC tool identifying two major parts: (1) external components and tools, and (2) internal components we implemented.

4.1. External Components

Ecore-Based Editor: Realizations can be written in any Ecore-based[10] modelling tool. Currently, we used Papyrus and Topcased open-source editors to create realizations of use case, activity, and component diagrams.

VML4RE and lightVC: We created VML4RE and a lightweight version of composition specification editor for architecture called *lightVC* (Lightweight Variability Composer). These languages use EMFTEXT[11] which provides the software infrastructure to derive a concrete syntax and plug-in based on the metamodel of each language written in Ecore. This technology separates concrete syntax and abstract syntax, easing maintenance of the languages. The concrete syntax chosen for VML4RE is HUTN (Human Usable Notation)[12] provided by EMFTEXT. In comparison with our previous tool version [5], HUTN allows a more usable and suitable notation for describing requirements composition.

SPLOT Editor and Parser: SPLOT allows us to do several activities such as to share and edit our models collaboratively via web, write arbitrary cross-tree constraints between features and generate an initial CNF formula of the feature models and its constraints. However, it is possible to use other feature model editors that translate to CNF or to implement the translation based on the mapping patterns described in Table 1.

[9] A prototype can be found at: http://www.mauricioalferez.com/JSS/JSS-Data.htm
[10] http://www.eclipse.org/modeling/emf/
[11] http://www.emftext.org/
[12] http://www.omg.org/spec/HUTN/

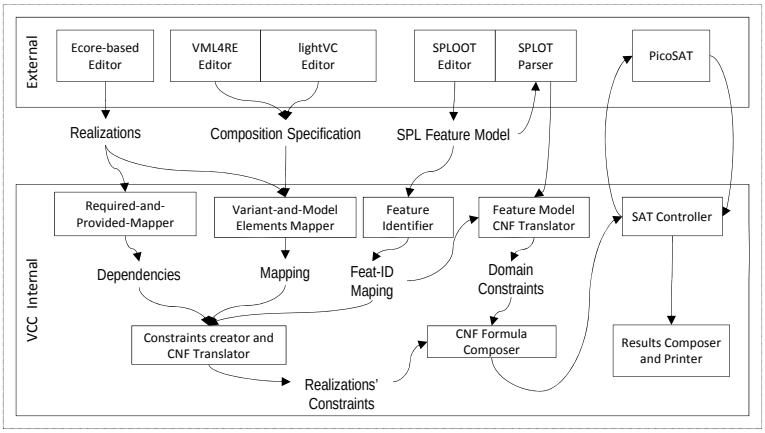


Figure 10: VCC tool support high-level architectural view.

PicoSAT: We employed PicoSat[13][14] as SAT solver to determine the satisfiability of each formula (we have also experimented SAT4J[15]).

4.2. VCC-Internal Components

Required-and-Provided Mapper: We have two implementations of this component. For requirements models, such as use case and activity diagrams, we obtain an initial list of required and provided elements based on the composition specification and realizations. For architectural models, we parse the component diagram to obtain the set of provided and required interfaces of each component.

Variant-and-Model Elements Mapper: It parses the composition specification to find the variants and the model elements that each one introduces into the use cases, activity and component diagrams.

Features' Identifier: It assigns and persists identifiers for each feature. Identifiers are used as variable names when creating constraints in CNF format and during satisfiability checking.

Constraints creator and CNF Translator: Given the dependencies between model elements and their relationships with variants, it deduces the realization constrains as described in Section 3.5. In the case of VML4RE and lightVC which use feature expressions written as propositional formula in prefix notation, it is necessary to translate to CNF, a format readable by SAT solvers.

Feature Model CNF Translator: Translates the clauses generated by SPLOT to an appropriate CNF form readable by SAT solvers [9].

CNF Formula Composer: Generates the formulas that will be passed to the SAT solver according to the patterns shown in Section 3.6, based on the constraints created by the *Feature Model CNF Translator* and *Constraints Creator and CNF Translator* components.

5. Evaluation

Experiments have been carried out to demonstrate the applicability of VCC for consistency checking in SPL systems (they are available online[16]).

5.1. Setup

We have evaluated VCC with two case studies. The first case study was the complete Smart Home SPL from which we used a small part as example in the first sections of this paper. The second case study is Mobile Photo [27] a SPL for applications that manipulates photos on mobile devices, such as mobile phones (there is another version called Mobile Media that includes more evolution releases). These case studies were selected because:

1. They are of different application domains while still be understood by readers in general given their everyday life utility.

[13] http://fmv.jku.at/picosat/

[14] We thanks to Alexander Nöhrer for its Java interface for PicoSAT

[15] http://sat4j.org/

[16] http://www.mauricioalferez.com/JSS/JSS-Data.htm

2. They use different kinds of features realizations: use scenarios for the Smart Home and component models for the Mobile Photo.
3. We had previous experience modelling variability and part of their realizations. The use scenarios in Smart Home were inspired on the requirements of the system [21] set by one of our industrial partners in the European Union project, AMPLE [28]. Together with Smart Home, Mobile Photo was used in the AMPLE project as an initial case study to research composition mechanisms in SPLs.

The objective of this evaluation was to determine if VCC can indicate where to find inconsistencies between features and different kinds of realizations. For the experiment, we defined feature models for each case study, realizations for each one, and the mapping between model elements and variants using VML4RE and lightVC. Next, we ran the VCC tool and analyzed the results.

Table 2 summarizes the rules that we used in this evaluation. We used 6 rules of type Requires and 4 of type Excludes, considering 4 kind of relationships between model elements.

#	Rule Description	Relationship
1	Required *inclusion Use Case* when *base Use Case* included	Inclusion
2	Required *Package* when any of its *Use Cases* are included	Containment
3	Req. *Package* when any of its contained *Packages* is included	Containment
4	Req. *Use Case* when any of its *children Use Cases* are included	Generalization
5	Req. *Actor* when at least one of its *children Actors* are included	Generalization
6	Req. *Activity* if at least one of its *Opaque Actions* are included	Containment
7	Required *required Interfaces* when *Component* is included	Usage/Realization
8	Excluded *base Use Case* when *inclusion Use Case* is excluded	Inclusion
9	Excluded *Use Case* when its *Package* is excluded	Containment
10	Excluded *provided Interfaces* when its *Component* is excluded	Usage/Realization
11	Excluded *Opaque Actions* when its *Activity* is excluded	Containment

Table 2: Summary of the rules implemented as realization constraints in our study

- Rules that consider INCLUDES relationships: an include relationship, in which one use case (the base use case) includes the functionality of another use case (the inclusion use case), supports the reuse of functionality in use-case diagrams. In VCC this kind of rule suggest including in a product the inclusion model element if the base model element is included.

- Rules that consider CONTAINMENT relationships: the contained model element is the one that requires a container. In UML a common container is the PACKAGE, but others exist, such as ACTIVITY that contains model elements represented in activity diagrams. In VCC this kind of rule suggest including in a product the container model element if the contained model element is included.

- Rules that consider the USAGE and REALIZATION relationships: USAGE relationship is a type of dependency relationship in which one model element (the client) requires another model element (the supplier) for full implementation or operation. In this experiment we consider USAGE and INTERFACE REALIZATION relationships in component diagrams employing components as clients, and interfaces as suppliers. In VCC this kind of rule suggests including in a product the supplier model element if the client model element is included.

- Rules that consider the GENERALIZATION relationships: a generalization relationship is a relationship in which one model element (the child) is based on another model element (the parent). Generalization relationships are used in class, component, deployment, and use case diagrams to indicate that the child receives all of the attributes, operations, and relationships that are defined in the parent. In our experiment we considered generalization relationships between ACTORS and between USE CASES. In VCC this kind of rule suggest including in a product the parent model element if the child model element is included.

Each rule instance requires at least two calls to the SAT solver, the first to check satisfiability of the left-hand side of the disjunction (i.e., $\neg(C_f \rightarrow FM_f)$) in Equation 1 and the second to check the right-hand side (i.e., $\neg(FM_f \rightarrow C_f)$).

Case Studies	Smart Home	Mobile Photo
Features	60	14
Main model elements	36 Use Cases, 12 Activities, 14 Packages.	12 Components, 15 Interfaces.
Variants	28	7
Rules	10	2
Rule instances checked	71	10
Product variants	One billion	16

Table 3: Evaluation results using VCC in the Smart Home SPL.

If VCC shows that it is satisfiable it means that there is an inconsistency. VCC provides information about the concrete variant, feature expression, action in the variant, and model elements in the realization that are related to the rule instance.

5.2. Results

Table 3 summarizes our evaluation for both cases studies[17]. Smart Home has 60 features and comprises significant aspects of modern home automation domain such as security, HVAC (Heating, Ventilating, and Air Conditioning), illumination control, fire control and multiple user interfaces. The Mobile Photo has 14 features and less variety and number of model elements than Smart Home. It has interconnected components, some acting as controllers for albums and photos and others implementing specific operations such as, view, create, delete, edit labels, and manage data access.

According to SPLOT feature analyzer, the Smart Home feature model allows the generation of ONE BILLION product configurations and sixteen are possible for Mobile Photo. These numbers show that manual consistency checking is not a good option; an approach and tool support such as the one that we proposed in this paper is very necessary.

In our experiments we found a total of 81 rules instances to check. The number of possible combinations of features, variants and model elements makes this task time consuming and error prone. The VCC approach and tool produces the results in the order of milliseconds when run on an Intel Core-Duo i5 at 2.4 Ghz. Given that feature models and constraints are mapped to clauses in VCC, the performance and scalability of our approach are proportional to the efficiency of the SAT solvers which are able to handle large number of clauses in industrial applications.

[17]The Smart Home was initially evaluated in a previous work [29] focusing only on consistency checking between features and use scenarios. In this evaluation we use extended equations for Excludes and Requires, and generalize VCC for any kind of realization.

6. Discussion and Related Work

An issue in the development of SPL is the lack of efficient approaches for consistency checking among models. In model-driven development this becomes an important issue as software is built by means of a chain of transformations. This can start from assets such as requirements specification models, to code-based assets that typically depend on a particular implementation technology. In this setting, the quality of the final code of target products depends mostly on (1) the transformations, (2) the source models of each transformation and (3) the information added after each transformation. Therefore, to create constraints not only helps stakeholders to understand the intended products, but also to obtain good quality source models that aim to derive good quality code.

We explore the use of SAT solvers and propositional logic to support consistency checking between a feature model and its corresponding model-based specifications. Usually SAT solvers and propositional formulas to represent dependencies between software assets are used much later in the development. The result of this work showed that they can be used much earlier and therefore some inconsistencies do not have to be left until later to be detected. The use of these methods is transparent for the SPL developer and therefore it does not add extra complexity to the modelling process. SAT solvers and the processing of propositional logic are implemented by libraries that are used internally by VCC.

Our work is related to previous work on type systems for SPL [11, 30, 12, 10, 13, 8]. Most of these works are based on feature-oriented programming where a feature is implemented by a code unit called *feature module*. When composed to a base program, a feature module introduces new structures, such as classes and methods, and refines existing ones. A program that is decomposed into features is called therefore, a feature-oriented program.

Thaker et al. address consistency checking as a *Safe Composition* problem [14, 31] for feature-oriented programs [13]. Safe Composition guar-

antees that all programs in a SPL are type safe, i.e., absent of references to undefined elements such as classes, methods, and variables. They show how safe composition properties can be verified for AHEAD SPL [32] using feature models and SAT solvers. Delaware et al. developed a formal model of the type system of Thaker et al. and proved its soundness [12].

Kastner and Apel provide a formal approach for type-checking of SPL systems [10]. They employ a calculus named *Color Featherweight Java* (CFJ) to develop an SPL of which variants can be generated as Featherweight Java (FJ) programs. They prove that given a well-typed CFJ SPL, all possible program variants that can be generated are well-typed FJ programs, i.e., generation preserves typing. Although this formalization covers only a small subset of Java, it provides theoretical insights about typing during the products generation mechanism.

Similarly to Thaker et al, [13] and, Kastner and Apel [10]., Apel et al. addresses feature-oriented programming, however, focusing on type checking independently of the target programming language and a subclass of possible type errors: *dangling references* [8]. In an SPL, code related to one feature may refer to code related to another feature, for example in the form of a method invocation or field access. If the former feature is selected and the latter is not, the former has a dangling reference, reported by the type system. *Feature Structure Trees* (FST) extended by *references* provide insight into the structural interactions. An FST organizes the feature module's structural elements (e.g., files, classes, fields, or methods) hierarchically while a reference is a pair of a name of source FST structural element and a name of a destination FST structural element. For a SPL to be checked, FSTs and references have to be extracted by language-specific code analysis tools. This approach was applied to two sample code-based SPLs written in Java and C. However, it provides two algorithms, *Global* and *Local Reference Checking* that are claimed to be independent of any programming technique. The two algorithms expect domain and structural information and provide information on dangling references and potential target features. Their difference is that the *Global* algorithm creates one propositional formula to be checked and the *Local* creates several formulas, each one describing the constraints implied by a different reference.

Similarly to Thaker et al., Kastner and Apel [10], and Apel et al. [8], VCC employs propositional formulas and SAT solvers to check its satisfiability. The *Local Reference Checking* algorithm of Apel et al. [8] is analogous in the way that VCC checks satisfiability because VCC generates one propositional formula by each consistency rule. However, in contrast to Apel et al., VCC does not use FST extended by references. VCC uses a *ModelElements-DependenciesModel* metaclass whose instances contain instances of the mappings *MapProvidedMe* and *MapRequiredMe* between model elements. Each model element can play several roles in the *MapProvidedMe* and *MapRequiredMe* mappings according to its relationships with other model elements: *requirer, requiredModelElement*, *provider* and *providedModelElement*. Thaker et al., and Apel et al., have *references* that are the analogous to the *MapRequiredMe* kind of mappings.

Another difference between VCC and Thaker et al., Kastner and Apel [10], and Apel et al. [8] is that it does not use feature modules but *Variant* modules (each one identified by a name and a feature expression). Therefore in VCC there is an M-to-N (where M, N >= 1) relationship between features and related model fragments. The implication of using variant modules instead of feature modules is that "individual features or small sets of related features typically change more or less independently from other features" [33], and thus a group of changes to code or models (such as the *Actions* in the *Variants* modules of VML4RE) can be related to threads drawn from a few related features. Particularly in domain specific application such as e-commerce systems, the ability to rapidly and consistently evolve sets of related features is key [33].

Even previously to the work of Thaker et al., Czarnecki and Pietroszek [30] presented an automatic verification procedure to ensure that no ill-structured template instances (i.e., concrete models of products) will be generated from a correct product configuration. Czarnecki and Pietroszek suggested the development of a type system that checks the entire assets of the feature-oriented SPL, instead of all individual feature-oriented programs. VCC as well as Thaker et al., Kastner and Apel follow that strategy and check consistency for the entire SPL assets instead of for individual products.

VCC complements the work of Czarnecki and Pietroszek with the fact that domain constraints obtained from the feature model could follow from constraints obtained from other model-based specifications. Therefore, this makes our general equa-

tion for consistency checking more complete. The implication of this fact is that when we check consistency between model-based specifications and features we do not assume that the feature model contains all domain constraints since its creation as it usually happens during early modelling in incremental SPL development. In fact, our approach benefits from the model-based specifications to suggest domain constraints in the feature model. Also, VCC does not rely on model templates annotated with feature expressions but on a separated composition model. However, using proper tool support it may be possible to obtain an instance of *ModelElementsDependenciesModel* based on the annotations and annotated model elements.

Previous work of Lopez-Herrejon and Egyed [34] also addressed consistency with an extension for *multi-view modelling* (MVM). Similar to the approaches mentioned in the previous parragraphs [10, 8, 13], a feature module captures the multiple *views* of a feature. Each view is represented by a different type of model, from which the authors focus on artefacts closer to software implementation, such as class, sequence and state diagrams. Lopez-Herrejon and Egyed observed that their approach for feature-oriented software development composes elements such as methods or classes (coarser granularity) but not their nested elements (finer granularity). VCC builds upon this previous work to take into account more than one type of model as well, and addressing models used in early SPL development. However, VCC proposes a metamodel and guidance on how to obtain instances of their metaclasses based on model-based specifications. The VCC metamodel makes explicit the relationships of key elements in consistency checking such as variants, feature expressions and the actions. Lopez-Herrejon and Egyed [34] use a single feature to identify a set of related model fragments of different type. Also, they employ a specific model transformation technique, called FOSD composition based on merging of feature modules. VCC considers more types of model transformations (*Actions*), such as connections and insertions of model elements and use them to obtain dependencies between model elements and mappings between variant modules (each variant module related to a feature expression) and model elements.

Another difference between VCC and other type systems such as the approach of Lopez-Herrejon and Egyed is the way they support modularization of the model or code fragments. In feature-oriented programming and development the modularization is mainly based on feature modules, however, VCC can support a dominant modularization different to features modules. For example, we employ a core use case diagram to represent a use case model for the entire system, instead of modelling each part of the use case diagram fragmented by feature modules. However, feature modules can be checked in VCC when each variant is related to a feature expression that contains only one feature (i.e., an instance of the metaclass *FeatureName* of type *Feature Expression*), and related to a set of model fragments that are related exclusively to that feature as possible. We do not speculate about which modularization technique for the model-based specifications is better. We just argue that more freedom to the developers should be provided to decide how their models should be modularized and composed.

There are other research areas related to our work but designed for single systems and designed for code-based assets. For example, SafeGen is a meta-programming language for writing statically safe generators of Java programs [31]. If a program generator written in SafeGen passes the checks of the SafeGen compiler, then the generator will only generate well-formed Java programs, for any generator input. SafeGen's static checking algorithm is a combination of traditional type checking for Java, and a series of calls to a theorem prover to check the validity of first-order logical sentences constructed to represent well-formedness properties of the generated program under all inputs.

Still for single systems but related to model-based specifications, authors such as Egyed considers well-formedness of UML based diagrams [35] and Jacobson and Ng consider aspect-oriented use cases diagrams [36]. These works do not check consistency of SPL models, and their composition mechanism does not support model weaving of model fragments as it is possible with a composition language. However, key ideas from these approaches may benefit consistency checking in SPL. For example, the knowledge about consistency rules between model elements of Egyed [35] can inspire the creation of consistency rules for SPL. Also, the idea of Jacobson [36] to modularize a system by use case packages, each one including all sort of model types that specify and implement an use case, is similar to the notion of feature modules and may be a starting point to consider early modelling in SPL development.

Other authors have worked on checking temporal properties in behavioural model-based specifi-

cations [37, 38]. This approach requires translating the SPL specifications to transition systems that explicitly relate individual transitions to atomic feature expressions. In comparison to that work, VCC is focused on checking structural properties, does not require from the developer to create other models such as transition systems, employs atomic and composed feature expressions, and it is not limited to negative variability composition mechanisms.

The approaches presented here are the ones closer to our work and a source of inspiration. However, there is a plethora of approaches related to software verification, consistency checking, safe composition, software testing and quality assurance that can be considered as related work but that cannot be examined in details. These cover other development paradigms [39], modelling techniques [40] and application domains [41, 42, 43]. Furthermore, there is a Common Variability Language (CVL) standardization initiative[18] of the OMG which is related to our approach because it also aims at expressing SPL variability and concrete changes in models according to the selection of features in a feature model. However, CVL does not propose how to check consistency between different model-based specifications and feature model.

7. Conclusions and Future Work

This paper defines constraints and presents tool support for consistency checking between features models and its model-based realizations in early SPL development. The aim of checking consistency is ensuring that the models for specific products derived from a feature model are consistent between them. VCC has been applied to requirements analysis and architectural models but it may be applied for other kind of models. The feasibility of VCC and its tool was evaluated with two case studies where the results showed that performance is not an issue.

As part of our future work in consistency checking, we will address the aspect of co-evolution of the models, it means to understand how the changes on each model (i.e., feature model, realizations and composition specification), influence its consistency with respect to the others. Also, as part of future work we will establish consistency rules and constraints for realizations that were not explicitly addressed in this paper, such as code modules or quality models. Here, we are addressing part of the problem in general.

[18] http://variabilitymodeling.org.

Appendix

In this Section we include the complete derivation of the requires and excludes rules. We use the following logical equivalences to translate expressions: A) $\neg(x \rightarrow y) \equiv x \wedge \neg y$, B) $x \rightarrow y \equiv \neg x \vee y$, and C) $\neg(z \vee w) \equiv \neg z \wedge \neg w$. At the right side of each line appears the letter that identifies each logical equivalence.

Requires

The requires constraint in Equation 6 can be obtained from a disjunction of Equation 8 and Equation 9:

$$
\begin{array}{ll}
\neg\left(C_{f,\,Req} \rightarrow FM_f\right) & \\
\equiv \neg\left(\left(F\left(Var\right) \rightarrow \bigvee_{i=1}^{n} F\left(VarReq_i\right)\right) \rightarrow FM_f\right) & \\
\equiv \left(F\left(Var\right) \rightarrow \bigvee_{i=1}^{n} F\left(VarReq_i\right)\right) \wedge \neg FM_f & (applying\ A) \\
\equiv \left(\neg F\left(Var\right) \bigvee_{i=1}^{n} F\left(VarReq_i\right)\right) \wedge \neg FM_f & (applying\ B)
\end{array}
\tag{8}
$$

$$
\begin{array}{ll}
\neg\left(FM_f \rightarrow C_{f,\,Req}\right) & \\
\equiv \neg\left(FM_f \rightarrow \left(F\left(Var\right) \rightarrow \bigvee_{i=1}^{n} F\left(VarReq_i\right)\right)\right) & \\
\equiv FM_f \wedge \neg\left(F\left(Var\right) \rightarrow \bigvee_{i=1}^{n} F\left(VarReq_i\right)\right) & (applying\ A) \\
\equiv FM_f \wedge F\left(Var\right) \wedge \neg \bigvee_{i=1}^{n} F\left(VarReq_i\right) & (applying\ A) \\
\equiv FM_f \wedge F\left(Var\right) \wedge \bigwedge_{i=1}^{n} \neg F\left(VarReq_i\right) & (applying\ C)
\end{array}
\tag{9}
$$

Excludes

The Excludes constraint in Equation 7 can be obtained from a disjunction of Equation 10 and Equation 11:

$$
\begin{array}{ll}
\neg\left(C_{f,\,Exc} \rightarrow FM_f\right) & \\
\equiv \neg\left(\left(F\left(Var\right) \rightarrow \neg \bigvee_{i=1}^{n} F\left(VarExc_i\right)\right) \rightarrow FM_f\right) & \\
\equiv \neg\left(\left(\neg F\left(Var\right) \vee \neg \bigvee_{i=1}^{n} F\left(VarExc_i\right)\right) \rightarrow FM_f\right) & (applying\ B) \\
\equiv \left(\neg F\left(Var\right) \vee \neg \bigvee_{i=1}^{n} F\left(VarExc_i\right)\right) \wedge \neg FM_f & (applying\ A) \\
\equiv \left(\neg F\left(Var\right) \vee \bigwedge_{i=1}^{n} \neg F\left(VarExc_i\right)\right) \wedge \neg FM_f & (applying\ C)
\end{array}
\tag{10}
$$

$$
\begin{array}{ll}
\neg\left(FM_f \rightarrow C_{f,\,Exc}\right) & \\
\neg\left(FM_f \rightarrow \left(F\left(Var\right) \rightarrow \neg \bigvee_{i=1}^{n} F\left(VarExc_i\right)\right)\right) & \\
\neg\left(FM_f \rightarrow \left(\neg F\left(Var\right) \vee \neg \bigvee_{i=1}^{n} F\left(VarExc_i\right)\right)\right) & (applying\ B) \\
FM_f \wedge \neg\left(\neg F\left(Var\right) \vee \neg \bigvee_{i=1}^{n} F\left(VarExc_i\right)\right) & (applying\ A) \\
FM_f \wedge F\left(Var\right) \wedge \bigvee_{i=1}^{n} F\left(VarExc_i\right) & (applying\ C)
\end{array}
\tag{11}
$$

Acknowledgments

This work was partially supported by the Portuguese research centre CITI, under the grant

PEst-OE/EEI/UI0527/2011, Portugal, the European project AMPLE, contract IST-33710 and the Portuguese grant SFRH/BD/46194/2008 of Fundação para a Ciência e a Tecnologia, Portugal. It was also partially funded by the Austrian FWF under agreement P21321-N15 and Marie Curie Actions—IEF project number 254965.

References

[1] P. Clements, L. Northrop, Software Product Lines: Practices and Patterns, Addison-Wesley, Boston, MA, USA, 2002.

[2] K. Pohl, G. Böckle, F. J. v. d. Linden, Software Product Line Engineering: Foundations, Principles and Techniques, Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2005.

[3] K. Czarnecki, U. W. Eisenecker, Generative programming: methods, tools, and applications, ACM Press/Addison-Wesley Publishing Co., New York, NY, USA, 2000.

[4] F. Heidenreich, J. Kopcsek, C. Wende, Featuremapper: mapping features to models, in: Companion of the 30th Int. Conf. on Software Engineering, ICSE Companion '08, ACM, New York, NY, USA, 2008, pp. 943–944.

[5] S. Zschaler, P. Sánchez, J. Santos, M. Alférez, A. Rashid, L. Fuentes, A. Moreira, J. Araújo, U. Kulesza, Vml* - a family of languages for variability management in software product lines, in: SLE, 2009, pp. 82–102.

[6] K. Czarnecki, M. Antkiewicz, Mapping features to models: A template approach based on superimposed variants, in: GPCE, 2005, pp. 422–437.

[7] M. Alférez, U. Kulesza, A. Sousa, J. Santos, A. Moreira, J. Araújo, V. Amaral, A model-driven approach for software product lines requirements engineering, in: SEKE, 2008, pp. 779–784.

[8] S. Apel, W. Scholz, C. Lengauer, C. Kästner, Language-independent reference checking in software product lines, in: FOSD, 2010, pp. 65–71.

[9] Int. confs. on theory and applications of satisfiability testing (2011).
URL `http://www.satisfiability.org/`

[10] C. Kästner, S. Apel, Type-checking software product lines - a formal approach, in: ASE, 2008, pp. 258–267.

[11] S. Apel, C. Kästner, A. Größlinger, C. Lengauer, Type safety for feature-oriented product lines, Autom. Softw. Eng. 17 (3) (2010) 251–300.

[12] B. Delaware, W. R. Cook, D. S. Batory, Fitting the pieces together: a machine-checked model of safe composition, in: ESEC/SIGSOFT FSE, 2009, pp. 243–252.

[13] S. Thaker, D. S. Batory, D. Kitchin, W. R. Cook, Safe composition of product lines, in: GPCE, 2007, pp. 95–104.

[14] S. S. Huang, D. Zook, Y. Smaragdakis, Statically safe program generation with safegen, in: GPCE, 2005, pp. 309–326.

[15] I. Jacobson, Object-Oriented Software Engineering: A Use Case Driven Approach, Addison Wesley Longman Publishing Co., Inc., Redwood City, CA, USA, 2004.

[16] P. Kruchten, The Rational Unified Process: An Introduction, 3rd Edition, Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2003.

[17] M. Alférez, J. Santos, A. Moreira, A. Garcia, U. Kulesza, J. Araújo, V. Amaral, Multi-view composition language for software product line requirements, in: SLE, 2009, pp. 103–122.

[18] F. Jouault, I. Kurtev, Transforming models with atl, in: MoDELS Satellite Events, 2005, pp. 128–138.

[19] G. Taentzer, Agg: A graph transformation environment for modeling and validation of software, in: AGTIVE, 2003, pp. 446–453.

[20] OMG, Meta object facility (mof) 2.0 query/view/transformation specification, Tech. rep., OMG (2008).
URL `http://www.omg.org/spec/QVT/1.0/PDF`

[21] H. Morganho, e. a. Gomes, Requirement specifications for industrial case studies, Deliverable D5.2, Ample Project (2008).
URL `www.ample-project.net`

[22] D. Benavides, S. Segura, A. R. Cortés, Automated analysis of feature models 20 years later: A literature review, Inf. Syst. 35 (6) (2010) 615–636.

[23] M. Mannion, Using first-order logic for product line model validation, in: SPLC, 2002, pp. 176–187.

[24] D. S. Batory, Feature models, grammars, and propositional formulas, in: SPLC, 2005, pp. 7–20.

[25] S. A. Cook, The complexity of theorem-proving procedures, in: STOC, 1971, pp. 151–158.

[26] A. Reder, A. Egyed, Model/analyzer: a tool for detecting, visualizing and fixing design errors in uml, in: ASE, 2010, pp. 347–348.

[27] E. Figueiredo, N. Cacho, C. Sant'Anna, M. Monteiro, U. Kulesza, A. Garcia, S. Soares, F. C. Ferrari, S. S. Khan, F. C. Filho, F. Dantas, Evolving software product lines with aspects: an empirical study on design stability, in: ICSE, 2008, pp. 261–270.

[28] AMPLE, Ample project (2009).
URL `http://www.ample-project.net`

[29] M. Alférez, R. E. Lopez-Herrejon, A. Moreira, V. Amaral, A. Egyed, Supporting consistency checking between features and software product line use scenarios, in: K. Schmid (Ed.), ICSR, Vol. 6727 of Lecture Notes in Computer Science, Springer, 2011, pp. 20–35.

[30] K. Czarnecki, K. Pietroszek, Verifying feature-based model templates against well-formedness ocl constraints, in: Proc. of the GPCE'06, GPCE '06, ACM, New York, NY, USA, 2006, pp. 211–220.

[31] S. S. Huang, D. Zook, Y. Smaragdakis, Statically safe program generation with safegen, Sci. Comput. Program. 76 (5) (2011) 376–391.

[32] D. S. Batory, Feature-oriented programming and the ahead tool suite, in: ICSE, 2004, pp. 702–703.

[33] M. L. Griss, Implementing product-line features by composing aspects, in: SPLC, 2000, pp. 271–289.

[34] R. E. Lopez-Herrejon, A. Egyed, Detecting inconsistencies in multi-view models with variability, in: ECMFA, 2010, pp. 217–232.

[35] A. Egyed, Fixing inconsistencies in uml design models, in: Proc. of the 29th Int. Conf. on Software Engineering, ICSE '07, IEEE Computer Society, Washington, DC, USA, 2007, pp. 292–301.

[36] I. Jacobson, P.-W. Ng, Aspect-Oriented Software Development with Use Cases (Addison-Wesley Object Technology Series), Addison-Wesley Professional, 2004.

[37] A. Classen, P. Heymans, P.-Y. Schobbens, A. Legay, J.-F. Raskin, Model checking lots of systems: efficient verification of temporal properties in software product

lines, in: ICSE (1), 2010, pp. 335–344.
[38] A. Classen, P. Heymans, P.-Y. Schobbens, A. Legay, Symbolic model checking of software product lines, in: ICSE, 2011, pp. 321–330.
[39] I. Schaefer, L. Bettini, F. Damiani, Compositional type-checking for delta-oriented programming, in: AOSD, 2011, pp. 43–56.
[40] G. Gröner, C. Wende, M. Boskovic, F. S. Parreiras, T. Walter, F. Heidenreich, D. Gasevic, S. Staab, Validation of families of business processes, in: CAiSE, 2011, pp. 551–565.
[41] W. E. Wong, W. K. Chan, T. H. Tse, F.-C. Kuo, Special issue on dynamic analysis and testing of embedded software, Journal of Systems and Software 85 (1) (2012) 1–2.
[42] D. Bucur, M. Z. Kwiatkowska, On software verification for sensor nodes, Journal of Systems and Software 84 (10) (2011) 1693–1707.
[43] A. Bertolino, S. Daoudagh, F. Lonetti, E. Marchetti, Automatic xacml requests generation for policy testing, in: ICST, 2012, pp. 842–849.